21世纪高等学校计算机专业实用规划教材

"好程序员成长"丛书



Python 快乐编程

基础入门

◎千锋教育高教产品研发部 / 编著

千锋教材定位——快乐学习，实战就业 。

免费提供一站式教学服务包，附赠配套的PPT、教学视频、教学大纲、考试系统、测试题等资源。

清華大學出版社

21 世纪高等学校计算机专业实用规划教材

“好程序员成长”丛书

Python 快乐编程基础入门

千锋教育高教产品研发部　编著

清华大学出版社
北京

内 容 简 介

本书致力于打造最适合 Python 初学者的入门教材，站在初学者角度，从零开始，由浅入深，以朴实生动的语言阐述复杂的问题，书中列举了大量现实中的例子进行讲解，同时搭配精心设计的插图，真正做到通俗易懂。本书共 14 章，涵盖 Python 基础语言、流程控制、基本数据类型、函数、模块与包、面向对象、文件、异常等核心知识点。每学完一个章节的知识点，便通过实用性强的案例，如“发红包”“扑克牌”“QQ 登录”等，将所学知识综合运用到实际开发中，积累项目开发经验。在每章末尾还配备了习题，用于对本章所学内容进行练习和巩固，达到即学即练的效果。

本书面向 Python 初学者、高等院校及培训学校的老师和学生，是牢固掌握 Python 语言开发技术的必读之作，同时也是通往深入探究人工智能的必经之路。

责任编辑： 贾　斌　李　晔
封面设计：
责任校对： 李建庄
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载： http://www.tup.com.cn，010-62795954
印 装 者：
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 17.5　　**字　　数：** 400 千字
版　　次： 2019 年 7 月第 1 版　　**印　　次：** 2019 年 7 月第 1 次印刷
印　　数： 1～　00
定　　价： 　元

产品编号：081392-01

本书编委会

（排名不分先后）

总　　监　胡耀文　古　晔

主　　编　杨　轩　潘松彪

副 主 编：彭晓宁　印　东　邵　斌

王琦晖　贾世祥　唐新亭

慈艳柯　朱丽娟　叶培顺

杨　斐　任条娟　舒振宇

卞秀运

序



为什么要写这样一本书

当今世界是知识爆炸的世界，科学技术与信息技术急速地发展，新型技术层出不穷。但教科书却不能将这些知识内容随时编入，致使教科书的有些知识内容显得陈旧不实用。在初学者还不会编写一行代码的情况下，就开始讲解算法，这样使初学者感到晦涩难懂，让初学者难以入门。

IT 行业，不仅仅需要理论知识，更需要实用型、技术过硬、综合能力强的人才。所以，高校毕业生求职面临的第一道门槛就是技能与经验的考验。学校往往注重学生的理论知识，忽略对学生的实践能力培养，因而导致学生无法将理论知识应用到实际工作中。

如何解决这一问题

为了解决这一问题，本书倡导的是快乐学习，实战就业。在语言描述上力求准确、通俗、易懂，在章节编排上力求循序渐进，在语法阐述时尽量避免术语和公式，从项目开发的实际需求入手，将理论知识与实际应用相结合。目标就是让初学者能够快速成长为初级程序员，并拥有一定的项目开发经验，从而在职场中拥有一个高起点。



千锋教育

前言 Foreword

在瞬息万变的 IT 时代，一群怀揣梦想的人创办了千锋教育，投身到 IT 培训行业。自 2011 年以来，一批批有志青年加入千锋教育，为了梦想笃定前行。千锋教育秉承用良心做教育的理念，为培养“顶级 IT 精英”而付出一切努力，为什么会有这样的梦想，我们先来听一听用人企业和求职者的心声：

“现在符合企业需求的 IT 技术人才非常紧缺，这方面的优秀人才我们会像珍宝一样对待，可为什么至今没有合格的人才出现？”

“面试的时候，用人企业问能做什么，这个项目如何来实现，需要多长的时间，我们当时都蒙了，回答不上来。”

“这已经是面试过的第十家公司了，如果再不行的话，是不是要考虑转行了，难道大学里的四年都白学了？”

“这已经是参加面试的第 N 个求职者了，为什么都是计算机专业毕业，但当问到项目如何实现时，却怎么连思路都没有呢？”

这些心声并非个别，而是现实社会中的普遍现象。高校的 IT 教育与企业的真实需求存在脱节，如果高校的相关课程仍然不进行更新的话，毕业生将面临难以就业的困境。很多用人单位表示，高校毕业生表象上知识丰富，但绝大多数在实际工作中用之甚少，甚至完全用不上高校学习阶段所学知识。针对上述存在的问题，国务院也做出了关于加快发展现代职业教育的决定。很庆幸，千锋教育所做的事情就是配合高校达成产学合作。

千锋教育致力于打造 IT 职业教育全产业链人才服务平台，在全国拥有数十家分校，数百名讲师，坚持以教学为本的方针，采用面对面教学，传授企业实用技能，教学大纲紧跟企业需求，拥有全国一体化的就业体系。千锋的价值观是“做真实的自己，用良心做教育”。

针对高校教师的服务：

1. 千锋教育基于近七年来的教育培训经验，精心设计了包含“教材+授课资源+考试系统+测试题+辅助案例”的教学资源包，节约教师的备课时间，缓解教师的教学压力，显著提高教学质量。

2. 本书配套代码和视频索取网址：http://www.codingke.com/。

3. 本书配备了千锋教育优秀讲师录制的教学视频，按本书知识结构体系部署到了教学辅助平台“扣丁学堂”上，可以作为教学资源使用，也可以作为备课参考。

高校教师如需索要配套教学资源，请关注“扣丁学堂”师资服务平台，扫描下方二维码关注微信公众号索取。



扣丁学堂

针对高校学生的服务：

1. 学 IT 有疑问，就找“千问千知”，它是一个有问必答的 IT 社区，平台上的专业答疑辅导老师承诺工作时间 3 小时内答复读者学习中遇到的专业问题。读者也可以通过扫描下方的二维码，关注千问千知微信公众号，浏览其他学习者在学习中分享的问题和收获。

2. 学习太枯燥，想了解其他学校的伙伴都是怎样学习的？你可以加入“扣丁俱乐部”。“扣丁俱乐部”是千锋教育联合各大校园发起的公益计划，专门面向对 IT 感兴趣的大学生提供免费的学习资源和问答服务，已有超过 30 多万名学习者从中获益。

就业难，难就业，千锋教育让就业不再难！



千问千知

关于本书

本书既可作为高等院校本、专科计算机相关专业的 Python 入门教材，还包含了千锋教育 Python 基础课程的全部内容，是一本适合广大计算机编程爱好者的优秀读物。

抢红包

本书配套源代码、习题答案的获取方法：添加小千 QQ 号或微信号 2133320438。

注意！小千会随时发放“助学金红包”。

致谢

千锋教育高教产品研发部组织编写了本书，将千锋 Python 课程多年积累的实战案例进行整合，通过反复精雕细琢最终完成了本书。另外，多名院校老师也参与了本书的部分编写与指导工作。除此之外，千锋教育 500 多名学员也参与到了教材的试读工作中，他们站在初学者的角度对教材提出了许多宝贵的修改意见，在此一并表示衷心的感谢。

意见反馈

在本书的编写过程中，虽然力求完美，但不足之处在所难免，欢迎各界专家和读者朋友给予宝贵意见，联系方式：huyaowen@1000phone.com。

千锋教育高教产品研发部

2018 年 7 月 25 日于北京

目录 Contents

学习Coding知识



获取配套教学资源包

考试系统 在线作业 云课堂

教学PPT 教学设计 ……

成就Coding梦想

在线视频：http://www.codingke.com/

配套源码：微信2570726663

　　　　　QQ 2570726663

学IT有疑问，就找千问千知！

第 1 章 chapter 1

Python 开发入门

本章学习目标

- 了解 Python 的特征与应用领域。
- 掌握 Python 的安装。
- 掌握 PyCharm 的安装与使用。

Python 语言诞生于 20 世纪 90 年代初，它从设计之初就秉承着简洁的宗旨。如今，Python 以其优美、清晰、简单的特性已成为全球最主流的编程语言之一。

1.1 Python 语言的简介

1.1.1 Python 语言的起源

Python 的创始人为 Guido van Rossum（荷兰人，见图 1.1）。1982 年，Guido 从阿姆斯特丹大学获得了数学和计算机硕士学位，由于当时的编程语言比较复杂，因此 Guido 希望能够研发出一种能够轻松编程的语言。ABC 语言（由荷兰的数学和计算机研究所开发）让 Guido 看到了希望，于是 Guido 应聘到该研究所工作，并参与到 ABC 语言的开发中。但由于当时的开发是单向的，因此最后只得到商业上失败的结果。



图 1.1 Python 之父

随着互联网的普及，Guido 再一次看到了希望。1989 年的圣诞节，这位宅男为了打发时间，决定在 ABC 语言的基础上开发一个新型的基于互联网社区的脚本解释程序，这样 Python 就在键盘敲击声中诞生了。Python 的诞生让 Guido 兴奋不已，但问题来了，这门新语言该用哪个名字来命名？某一天，Guido 在欣赏他最喜爱的喜剧团体 Monty Python 演出时，突然灵光一闪，这门新语言有了自己的命名——Python（大蟒蛇的意思）。

1.1.2 Python 语言的发展

Python 从诞生一直到现在，经历了多个版本。截止到目前，官网仍然保留的版本主要是基于 Python 2.x 和 Python 3.x 系列，具体如表 1.1 所示。

表 1.1 Python 版本及发布时间

版　本	时　间	版　本	时　间
Python 1.0	1994/01	Python 3.1	2009/06/27
Python 2.0	2000/10/16	Python 3.2	2011/02/20
Python 2.4	2004/11/30	Python 3.3	2012/09/29
Python 2.5	2006/09/19	Python 3.4	2014/03/16
Python 2.6	2008/10/01	Python 3.5	2015/09/13
Python 2.7	2010/07/03	Python 3.6	2016/12/23
Python 3.0	2008/12/03	Python 3.7	2018/06/27

Python 2.7 是 Python 2.x 系列的最后一个版本，已经停止开发，计划在 2020 年终止支持。Guido 决定清理 Python 2.x 系列，并将所有最新标准库的更新改进体现在 Python 3.x 系列中。Python 3.x 系列的一个最大改变就是使用 UTF-8 作为默认编码，从此，在 Python 3.x 系列中就可以直接编写中文程序了。

另外，Python 3.x 系列比 Python 2.x 系列更规范统一，其中去掉了某些不必要的关键字与语句。由于 Python 3.x 系列支持的库越来越多，开源项目支持 Python 3.x 的比例已大大提高。鉴于以上理由，本书推荐读者直接学习 Python 3.x 系列。

1.1.3 Python 语言的特征

1．简单

Python 是一种代表简单主义思想的语言，阅读一段 Python 程序就像在阅读一篇文章，这使开发者能够专注于解决问题而不是去搞明白语言本身。

2．易学

Python 有极其简单的语法，如果开发同样的功能，使用其他语言可能需要上百行代码，而 Python 只需几十行代码就可以轻松搞定。

3．免费、开源

Python 是 FLOSS（自由/开放源码软件）之一，使用者可以自由地发布这个软件的副本、阅读它的源代码并对它进行修改，这也是 Python 如此优秀的原因之一。

4．可移植性

由于其开源本质，Python 已经被移植在许多平台上，例如 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE 等。

5．解释性

C/C++语言在执行时需要经过编译，生成机器码后才能执行。Python 是直接由解释

器执行的。由于不再需要担心如何编译程序、如何确保连接装载正确的库等，所有这一切都使得 Python 的使用更加简单。

6. 面向对象

Python 从设计之初就已经是一门面向对象的语言。在面向过程的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在面向对象的语言中，程序是由数据和功能组合而成的对象构建起来的。

7. 可扩展性

假如用户需要一段关键代码运行得更快或者希望某些算法不公开，可以把部分程序用 C 或 C++语言编写，然后在 Python 程序中使用它们。

8. 可嵌入性

用户可以把 Python 嵌入到 C/C++程序，从而向程序提供脚本功能。

9. 丰富的库

Python 提供丰富的标准库，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI、Tk 以及其他与系统相关的库。

1.1.4 Python 语言的应用领域

1. Web 开发

Python 语言支持 Web 网站开发，比较流行的开发框架有 Flask、Django 等。许多大型网站就是用 Python 开发的，例如 YouTube、Google、金山在线、豆瓣等。

2. 网络爬虫

Python 语言提供了大量网络模块用于对网页内容进行读取和处理，如 urllib、cookielib、httplib、scrapy 等。同时，这些模块结合多线程编程以及其他有关模块可以快速开发网页爬虫之类的应用程序。

3. 科学计算与数据可视化

Python 语言提供了大量模块用于科学计算与数据可视化，如 NumPy、SciPy、SymPy、Matplotlib、Traits、TraitsUI、Chaco、TVTK、Mayavi、VPython、OpenCV 等，这些模块涉及的应用领域包括数值计算、符号计算、二维图表、三维数据可视化、三维动画演示、图像处理以及界面设计等。

此外，Python 语言在系统编程、GUI 编程、数据库应用、游戏、图像处理、人工智

能等领域被广泛应用。

1.2 Python 的安装

工欲善其事，必先利其器。在学习 Python 语言之前，首先要搭建 Python 开发环境，本书将基于 Windows 平台开发 Python 程序，接下来分步骤讲解 Python 的安装。

（1）在浏览器地址栏中输入“http://python.org/”，按回车键，进入 Python 官方网站，如图 1.2 所示。

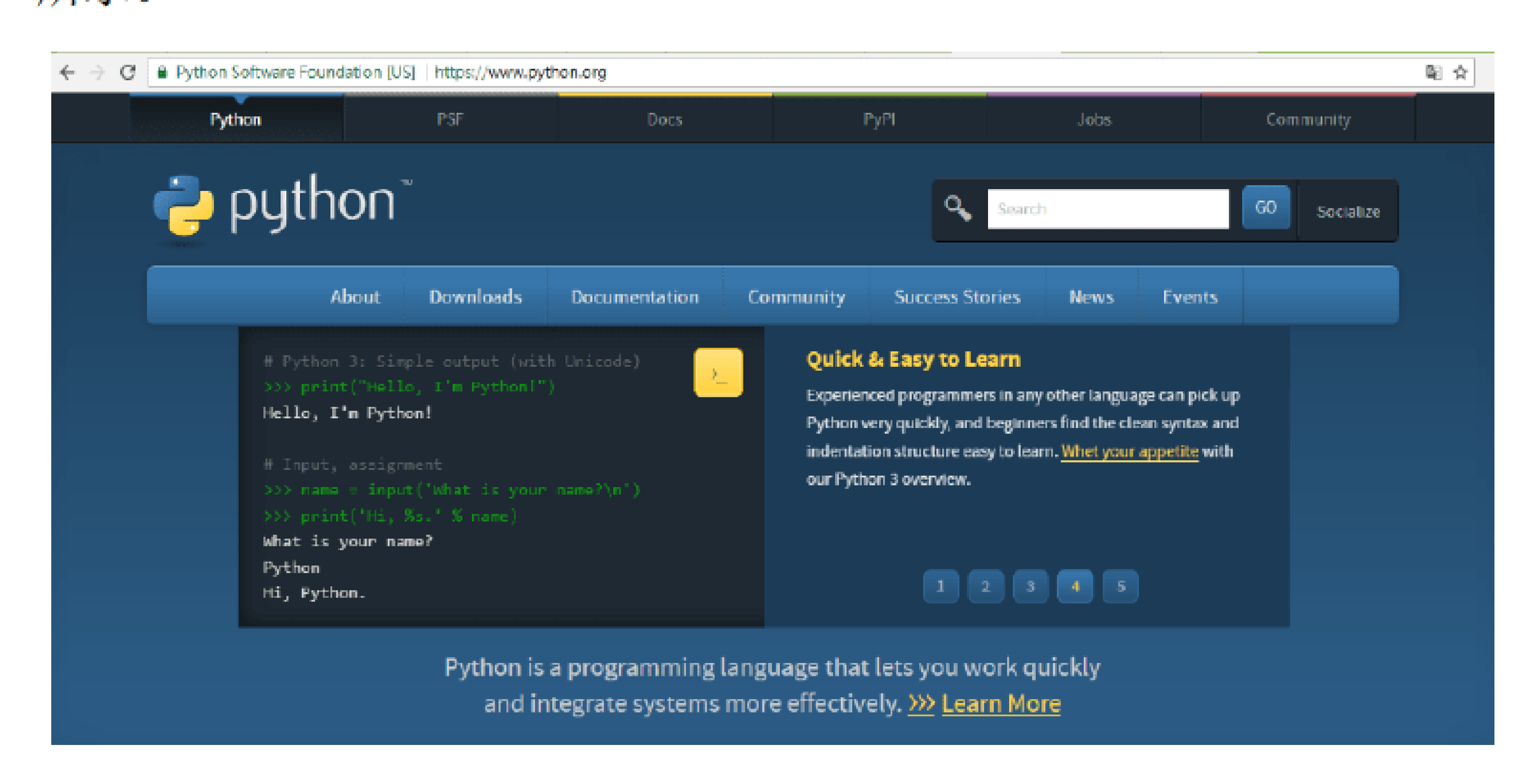


图 1.2　Python 官网

（2）单击图 1.2 中的 Downloads 按钮进入下载页面，如图 1.3 所示。

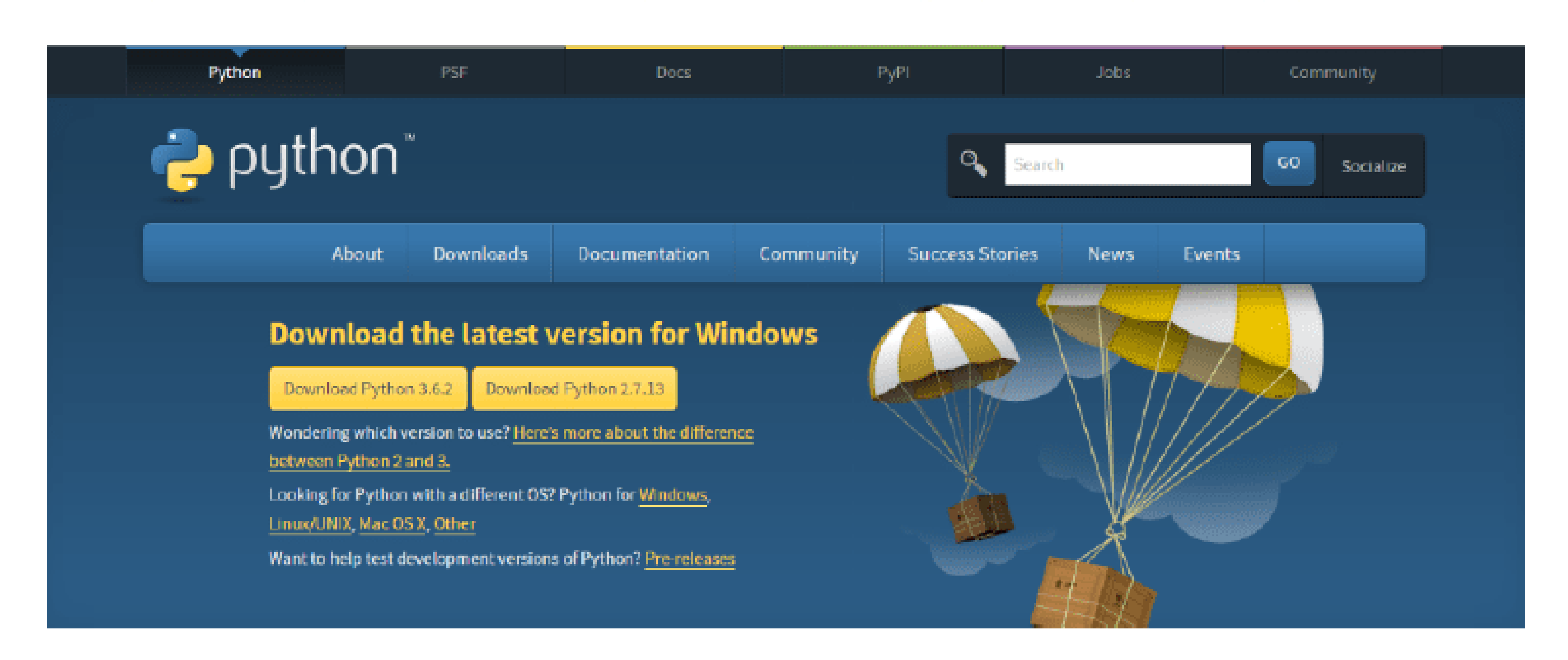


图 1.3　下载页面

（3）单击图 1.3 中的 Download Python 3.6.2 按钮进行下载，下载完成后的文件名为 python-3.6.2.exe，双击该文件，进入 Python 安装界面，如图 1.4 所示。

（4）在图 1.4 中，选中 Add Python 3.6 to PATH 选项，表示将 Python.exe 添加到环境变量 Path 中，此外还可以选择安装方式，Install Now 为默认安装，Customize installation 为自定义安装，此处单击 Customize installation 选项，进入可选特性界面，如图 1.5 所示。



图 1.4　安装界面

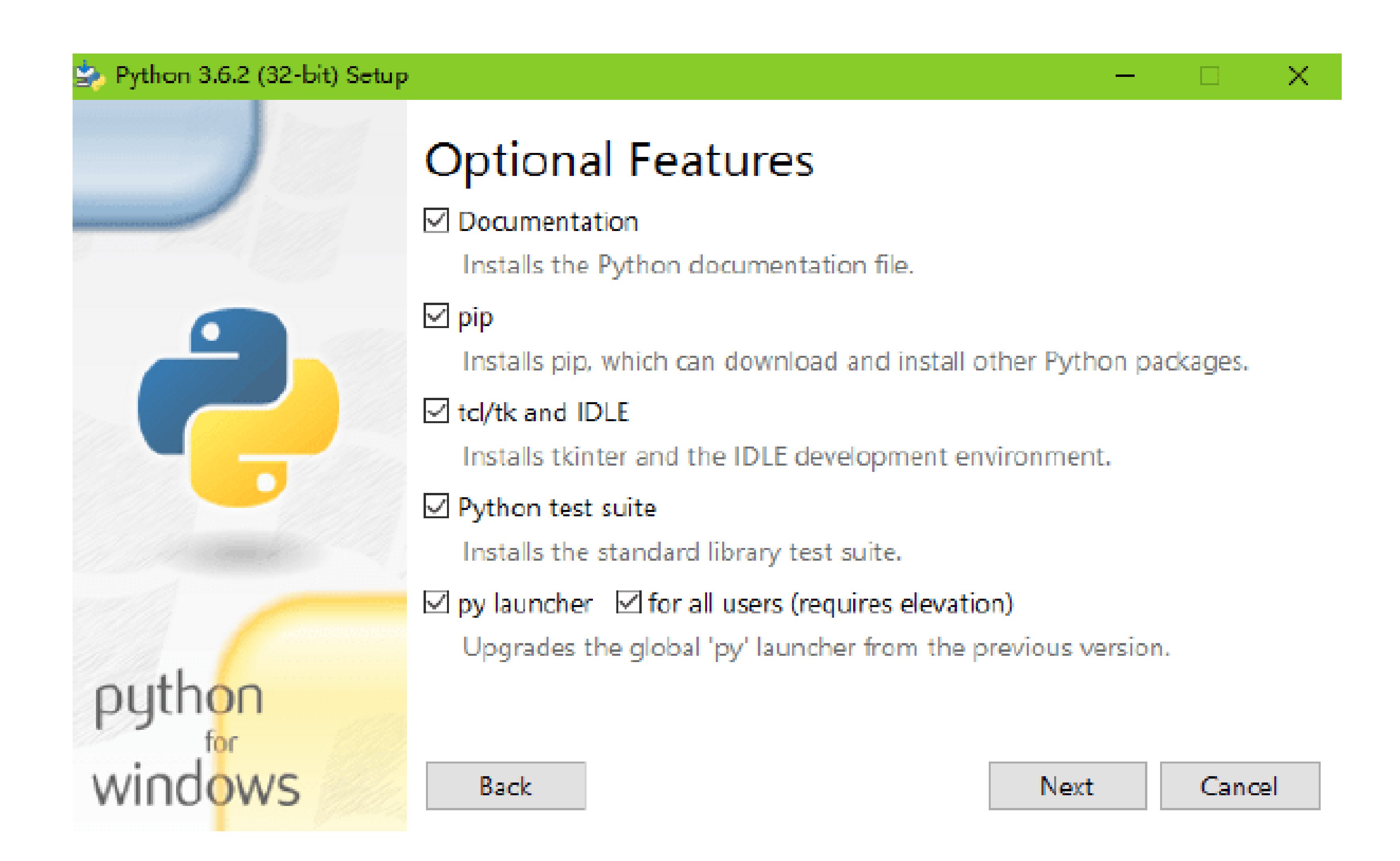


图 1.5　可选特性界面

（5）单击图 1.5 中的 Next 按钮，进入高级选项界面，如图 1.6 所示。

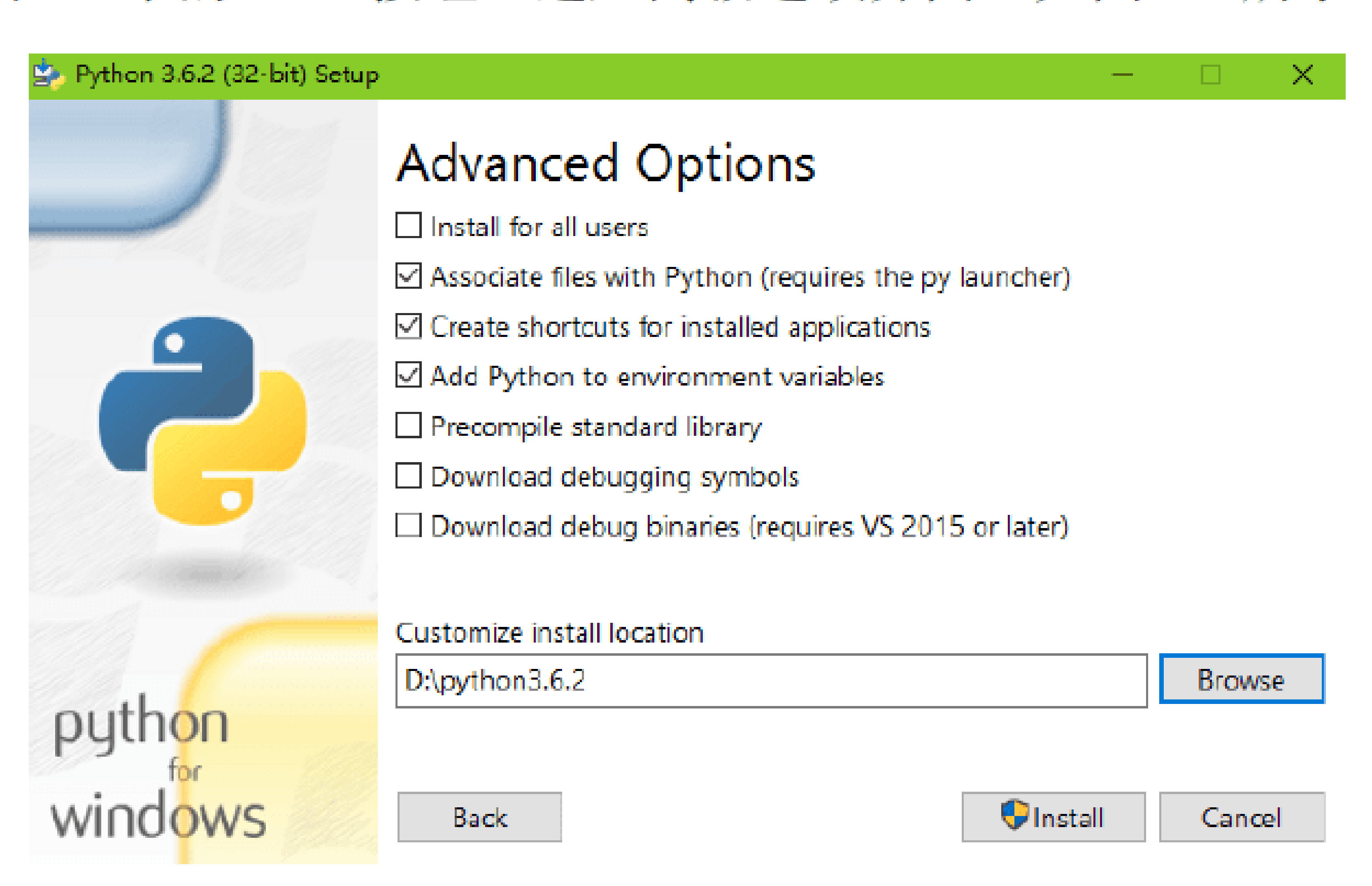


图 1.6　高级选项界面

（6）单击图 1.6 中的 Browse 按钮，选择安装路径，此处选择 D:\python3.6.2，最后

单击 Install 按钮，开始安装，进入安装进度界面，如图 1.7 所示。

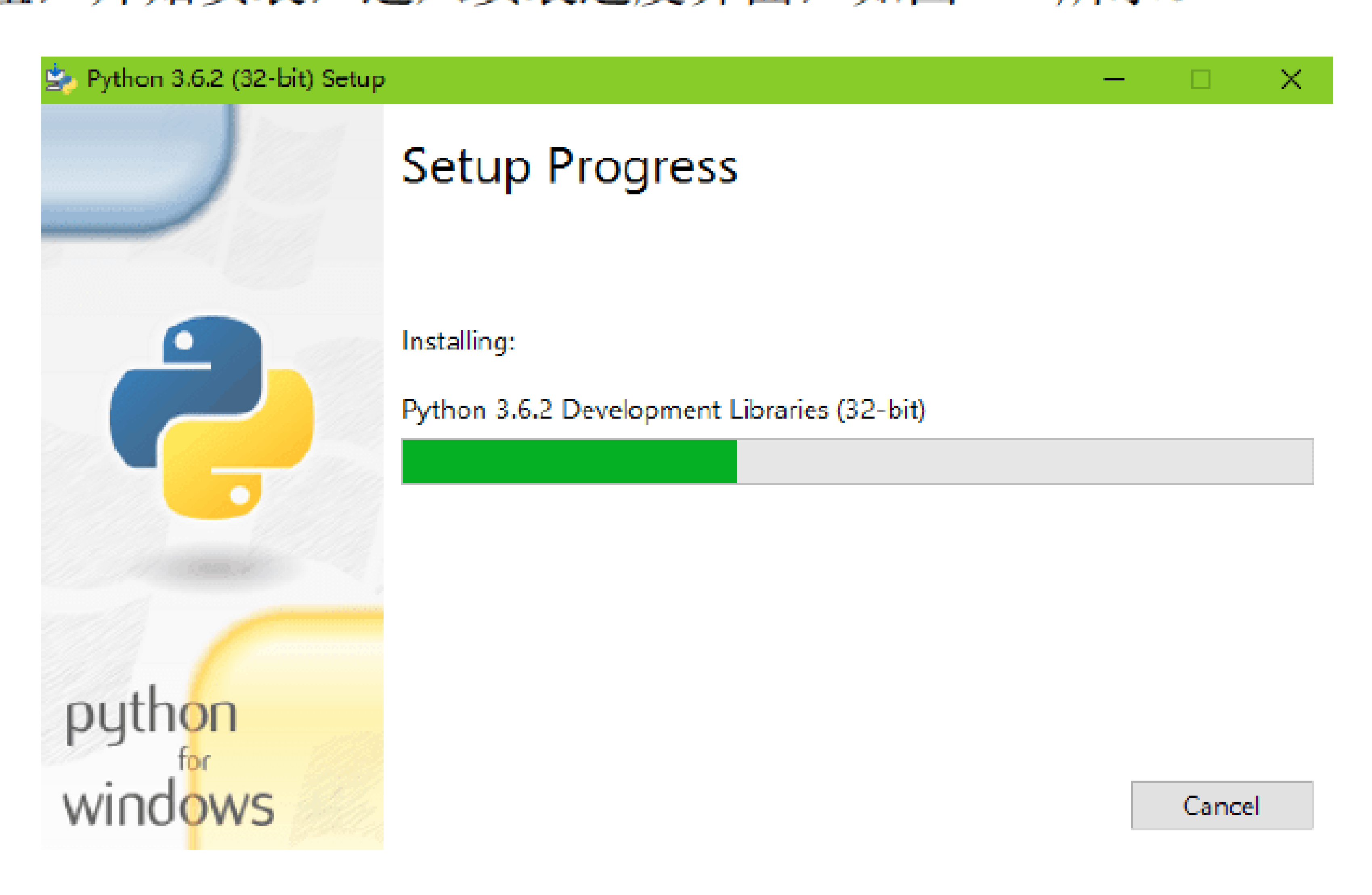


图 1.7　安装进度界面

（7）安装完成后的界面如图 1.8 所示，最后单击 Close 按钮即可。

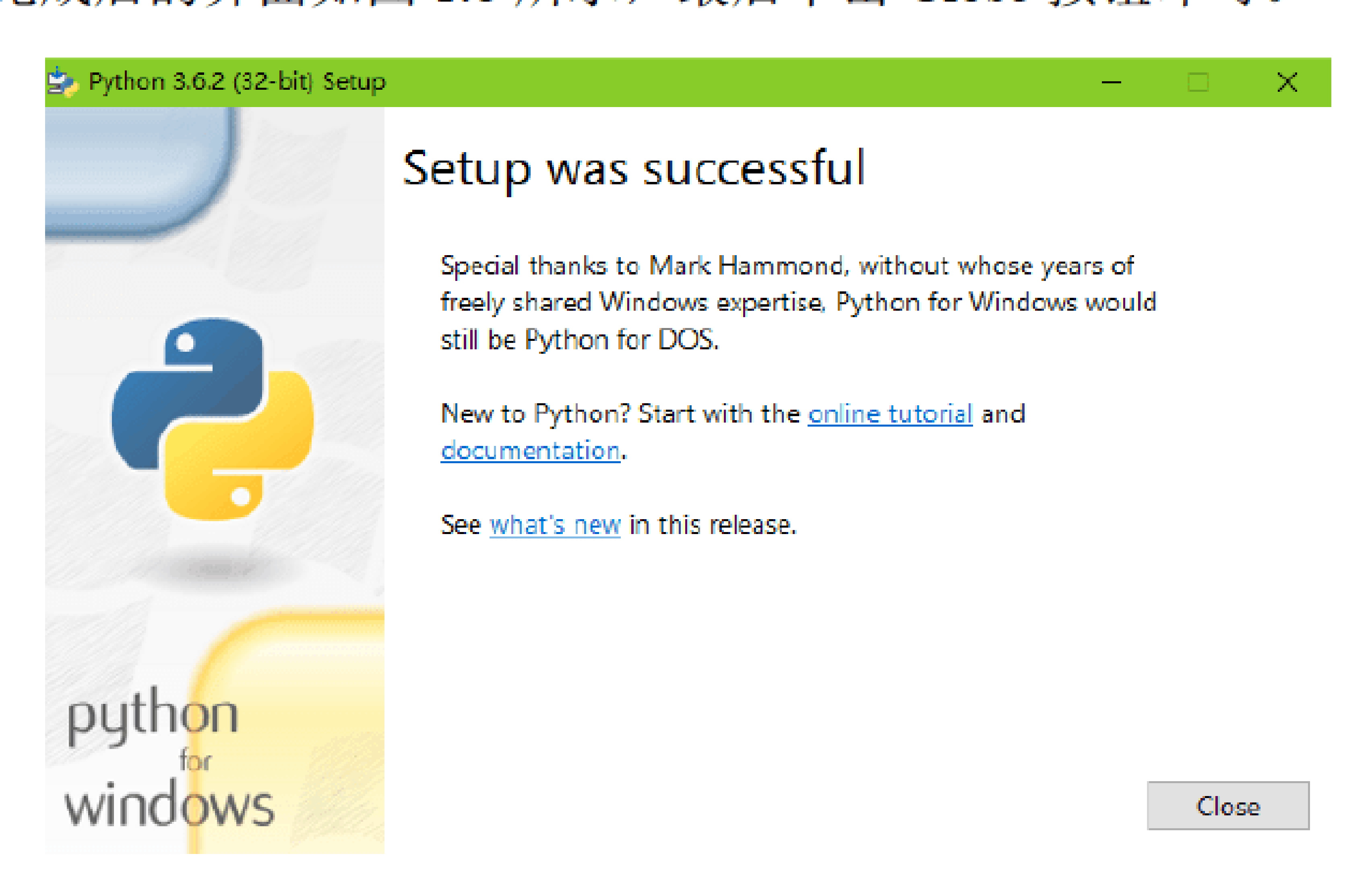


图 1.8　安装完成界面

（8）安装完成后，需要测试安装的 Python 是否可用。打开控制台（按 Window+R 组合键打开运行窗口，在输入框中输入 cmd 并单击“确定”按钮），在命令行中输入 python，按回车键，将会显示 Python 的版本号，如图 1.9 所示。



图 1.9　测试 Python 环境

在图 1.9 中，输入 python 并按回车键后，Python 解释器就开始启动了，用户可以接着输入“import this”，如图 1.10 所示。

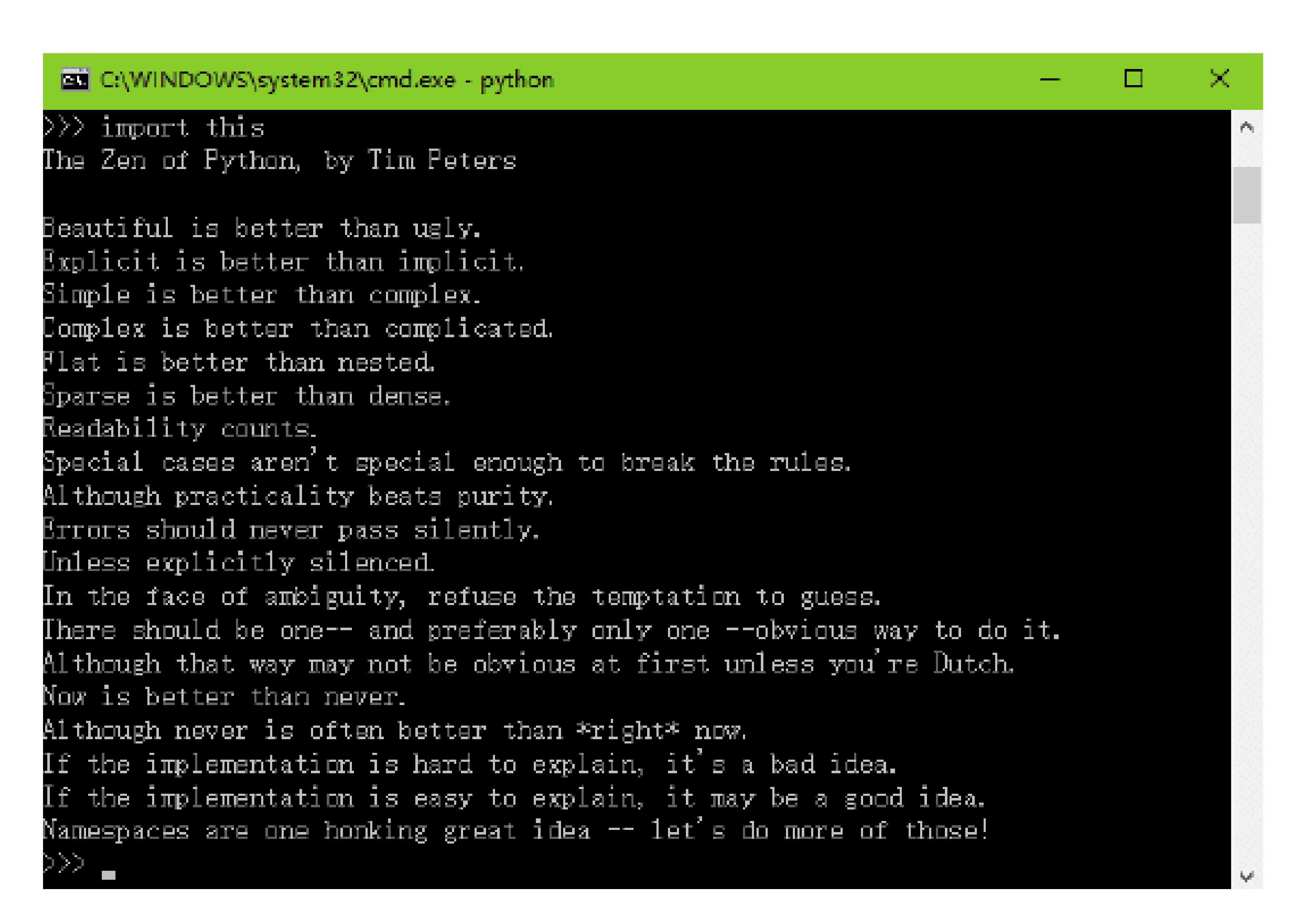


图 1.10　输入“import this”

在图 1.10 中，输出结果为 Python 的设计哲学，即优雅、明确、简单。如果想退出 Python 解释器，则输入 exit()。

1.3　集成开发环境 PyCharm

成功安装 Python 环境后，在控制台中是无法进行 Python 开发的，还需要安装一个专属工具来编写 Python 代码，即 PyCharm。它是一种 IDE（Integrated Development Environment，集成开发环境），具备语法高亮、调试、实时比较、Project 管理、代码跳转、智能提示、单元测试、版本控制等功能，可以很好地提高程序开发效率。

1.3.1　PyCharm 的安装

（1）打开 PyCharm 官方网站http://www.jetbrains.com/pycharm/，如图 1.11 所示。

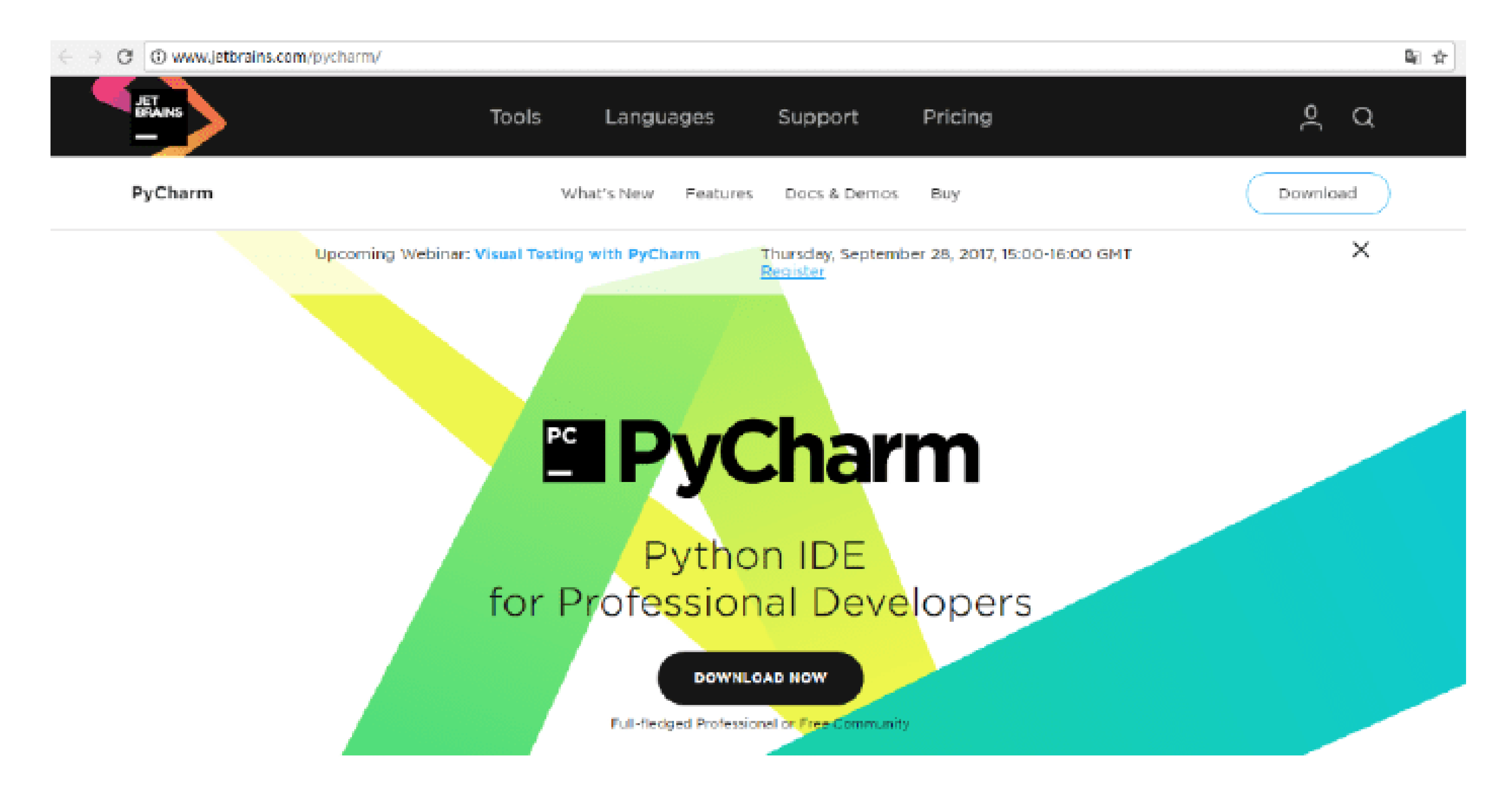


图 1.11　PyCharm 下载页面

（2）单击图 1.11 中的 DOWNLOAD NOW 按钮进入下载页面，如图 1.12 所示。

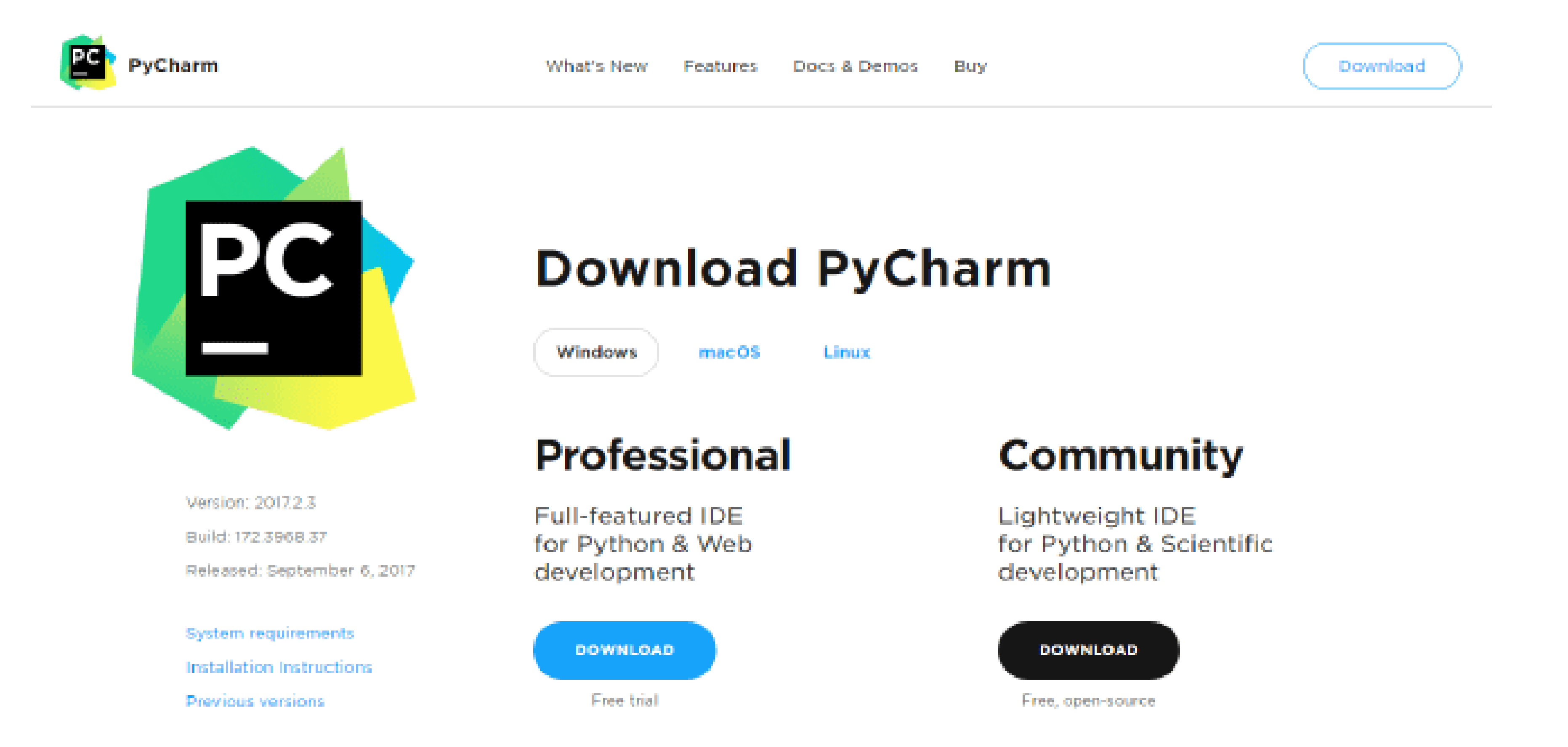


图 1.12　PyCharm 的版本

（3）单击图 1.12 中 Professional 版本下的 DOWNLOAD 按钮进行下载，下载完成后的文件名为 pycharm-professional-2017.2.3.exe，双击该文件，进入 PyCharm 安装界面，如图 1.13 所示。

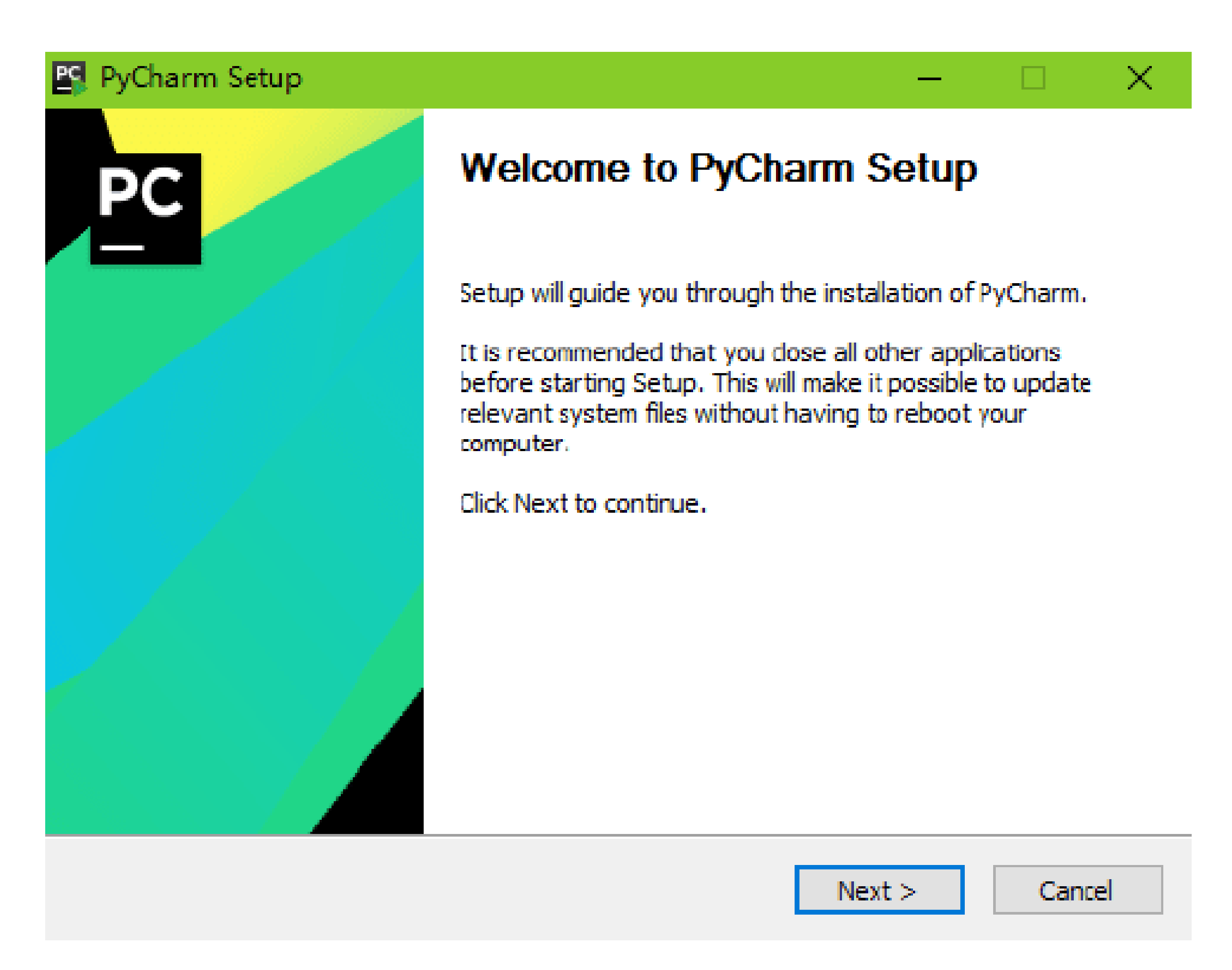


图 1.13　安装界面

（4）单击图 1.13 中的 Next 按钮，进入选择安装路径界面，如图 1.14 所示。

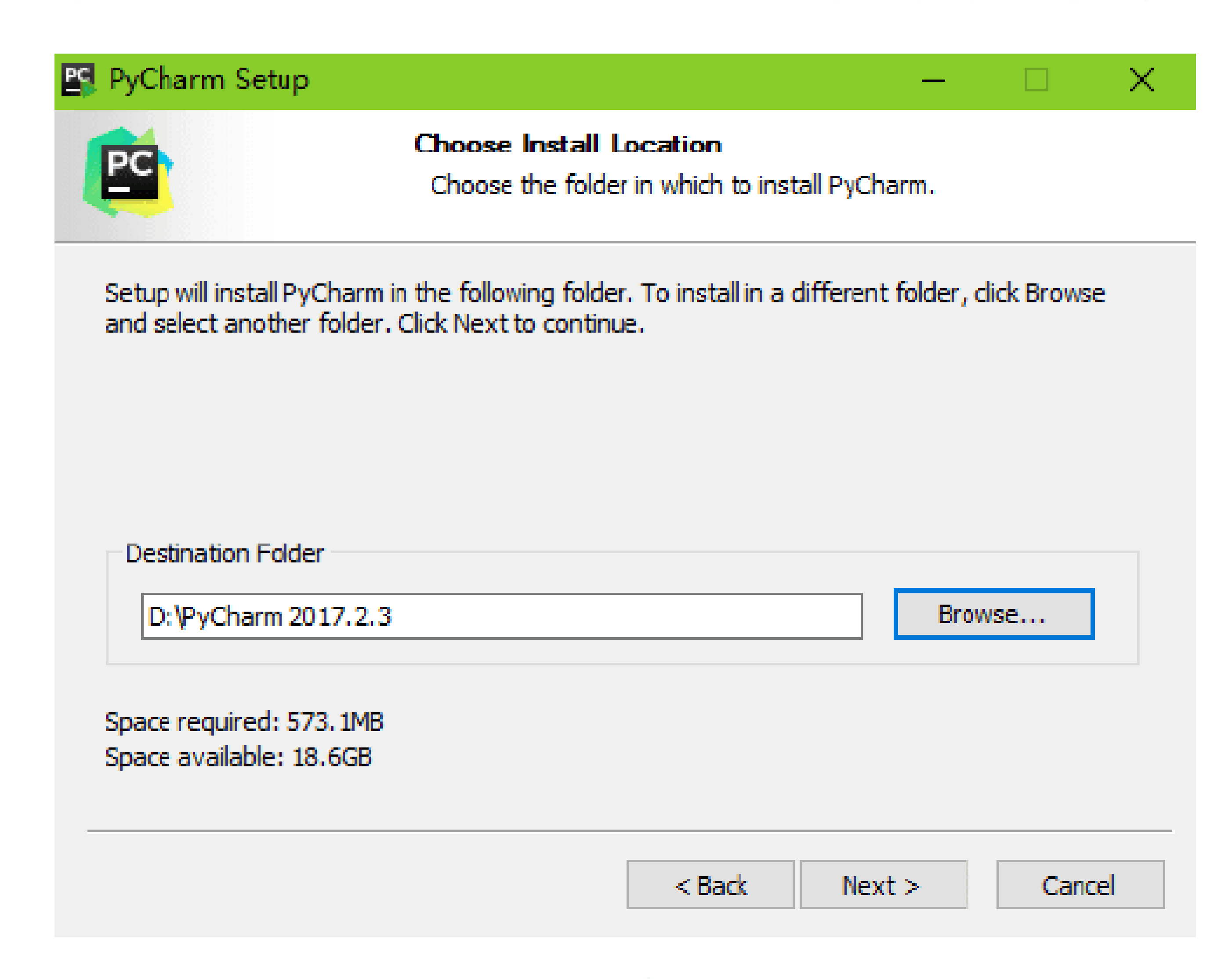


图 1.14　选择安装路径界面

（5）单击图 1.14 中的 Next 按钮，进入配置安装界面，如图 1.15 所示。

PyCharm Setup
Installation Options
Configure your PyCharm installation
Create Desktop shortcut
32-bit launcher
64-bit launcher
Create associations
.py
Download and install JRE x86 by JetBrains
< Back
Next >
Cancel

图 1.15　配置安装界面

（6）单击图 1.15 中的 Next 按钮，进入选择启动菜单界面，如图 1.16 所示。

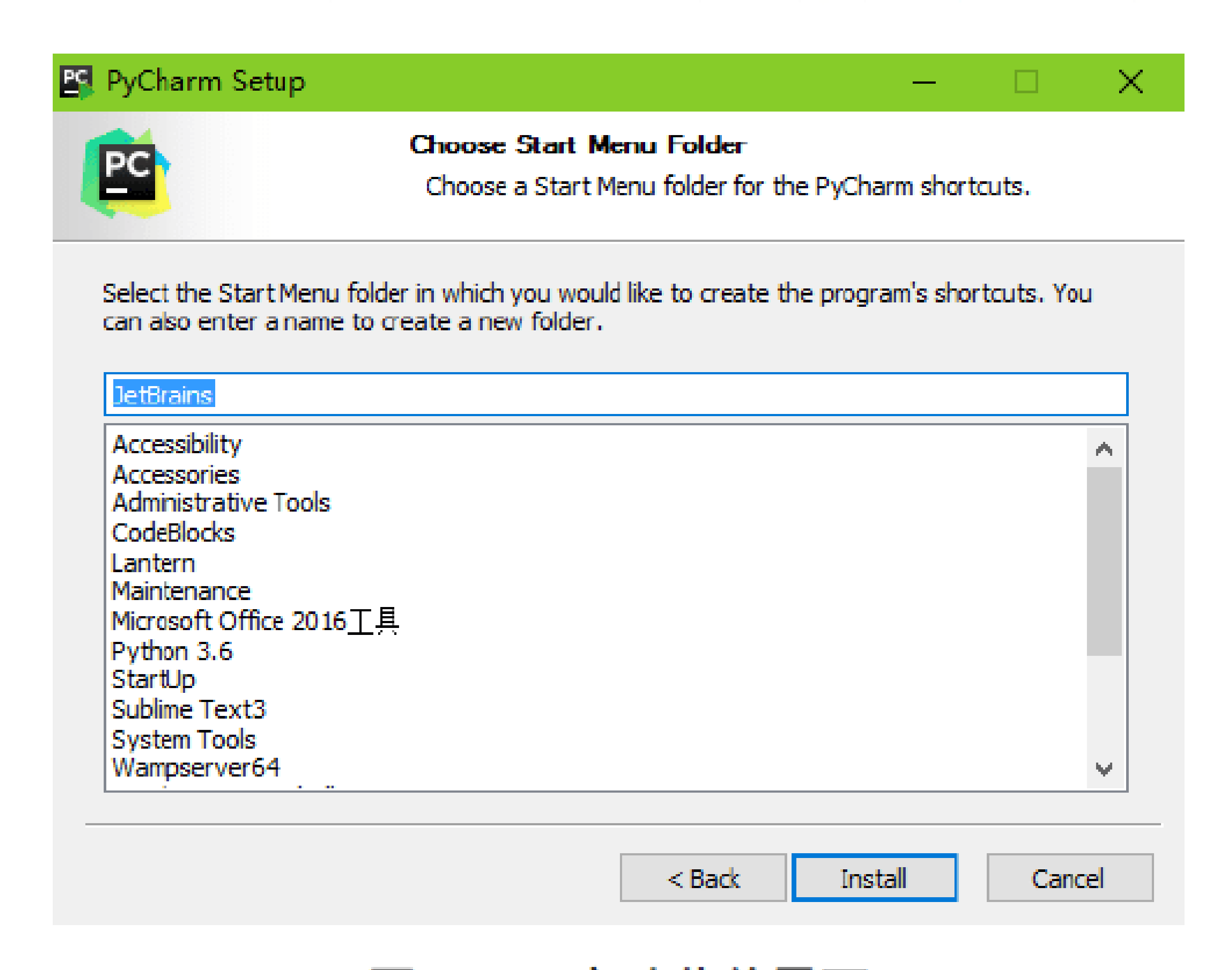


图 1.16　启动菜单界面

（7）单击图 1.16 中的 Install 按钮，进入安装过程界面，如图 1.17 所示。

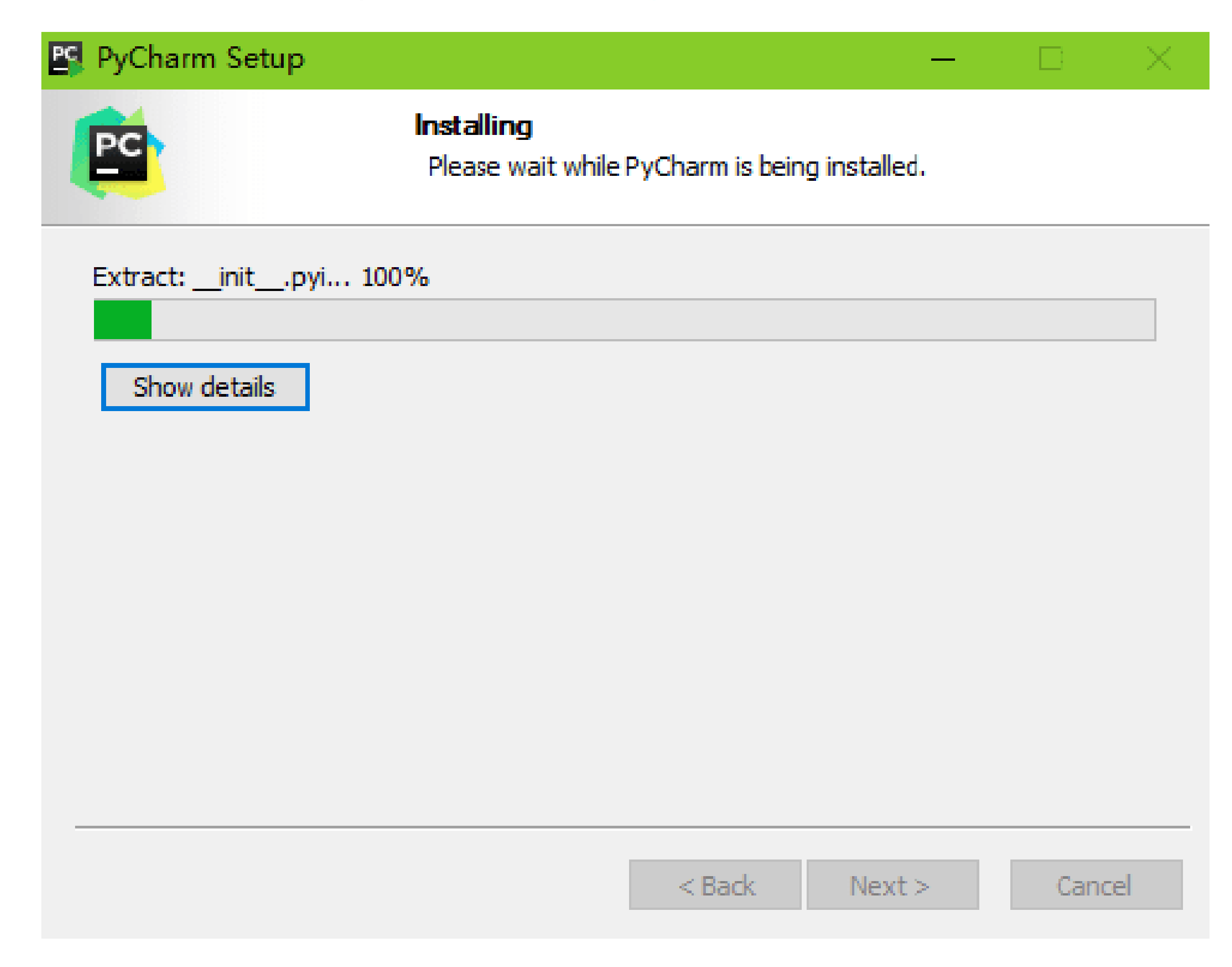


图 1.17　安装过程界面

（8）安装完成后的界面如图 1.18 所示，最后单击 Finish 按钮即可。

图 1.18　完成安装界面

1.3.2　PyCharm 的使用

（1）完成安装后，用户可以尝试使用 PyCharm。双击 PyCharm 的快捷方式运行程序，PyCharm 支持导入以前的设置，由于用户是初次使用，直接选择 Do not import settings 选项（不导入之前设置），如图 1.19 所示。

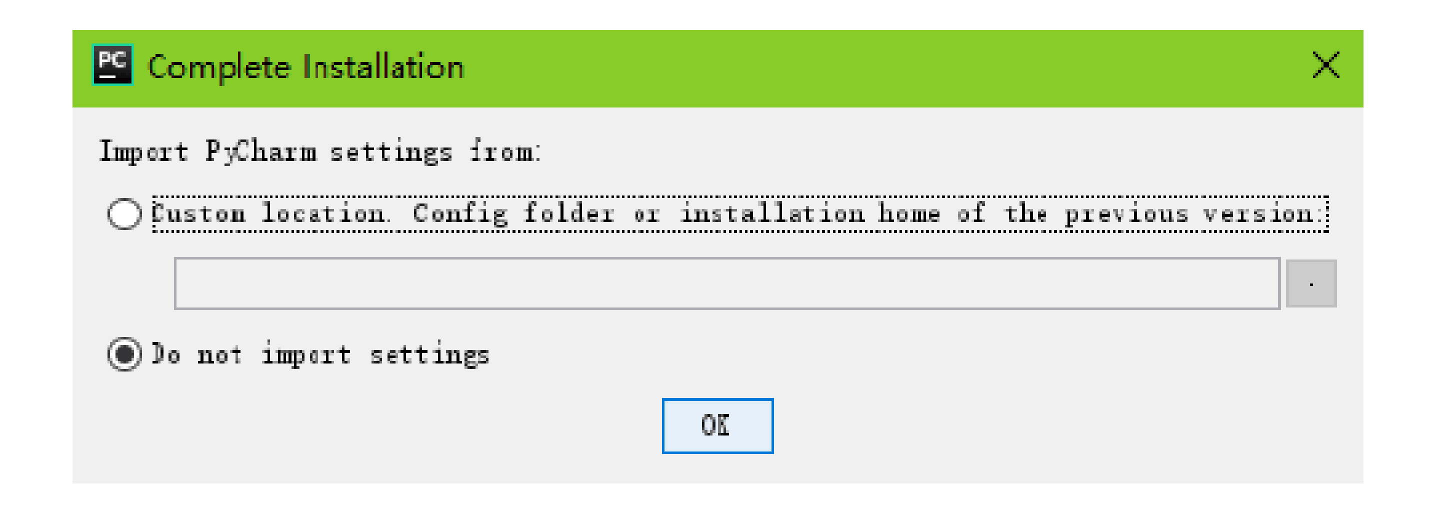


图 1.19　导入配置界面

（2）单击图 1.19 中的 OK 按钮，进入许可证激活界面，如图 1.20 所示。

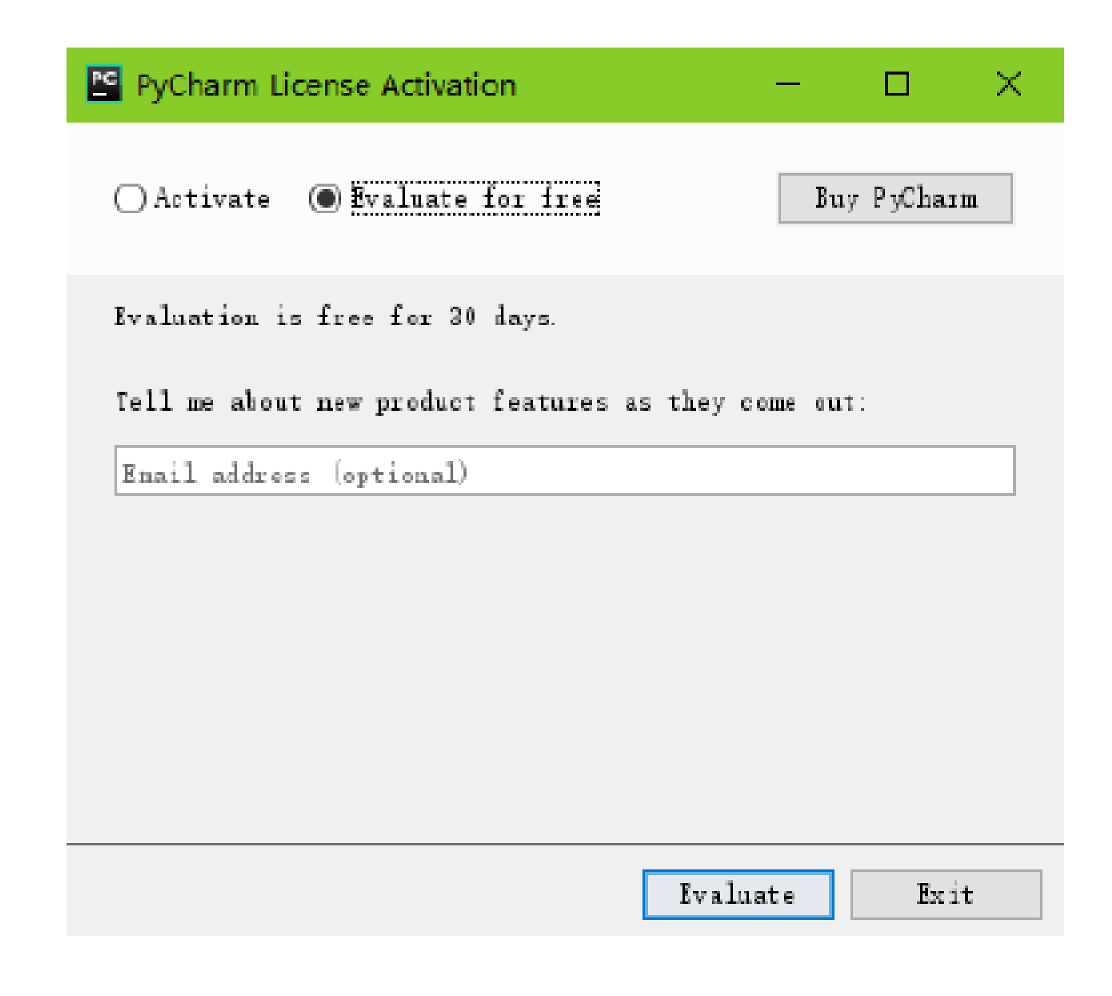


图 1.20　许可证激活界面

（3）选择图 1.20 中的 Evaluate for free 选项并单击 Evaluate 按钮，进入提示用户协议界面，如图 1.21 所示。

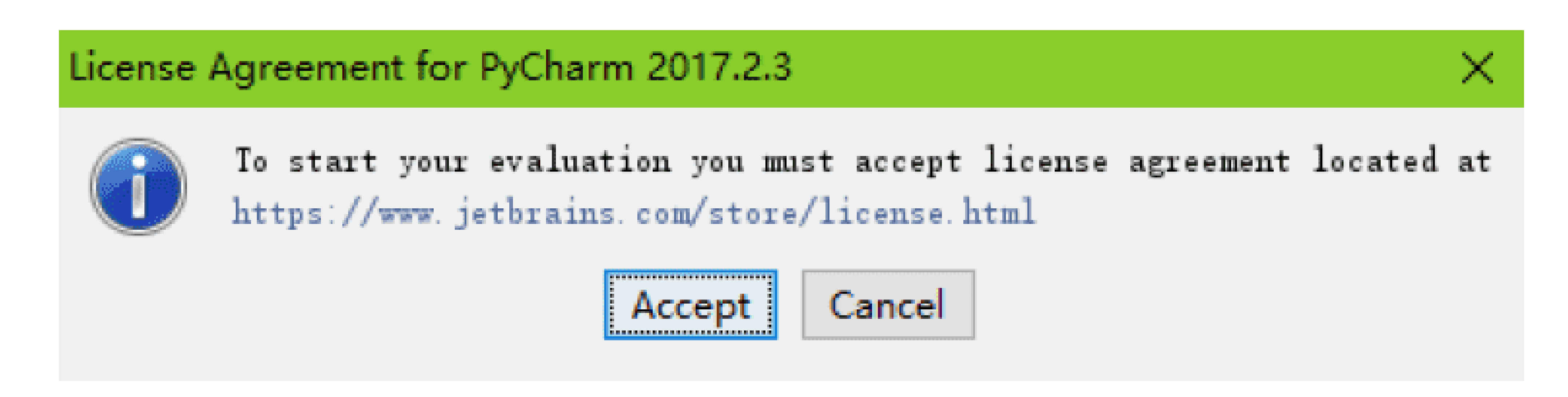


图 1.21 用户协议界面

（4）单击图 1.21 中的 Accept 按钮，进入启动界面，如图 1.22 所示。

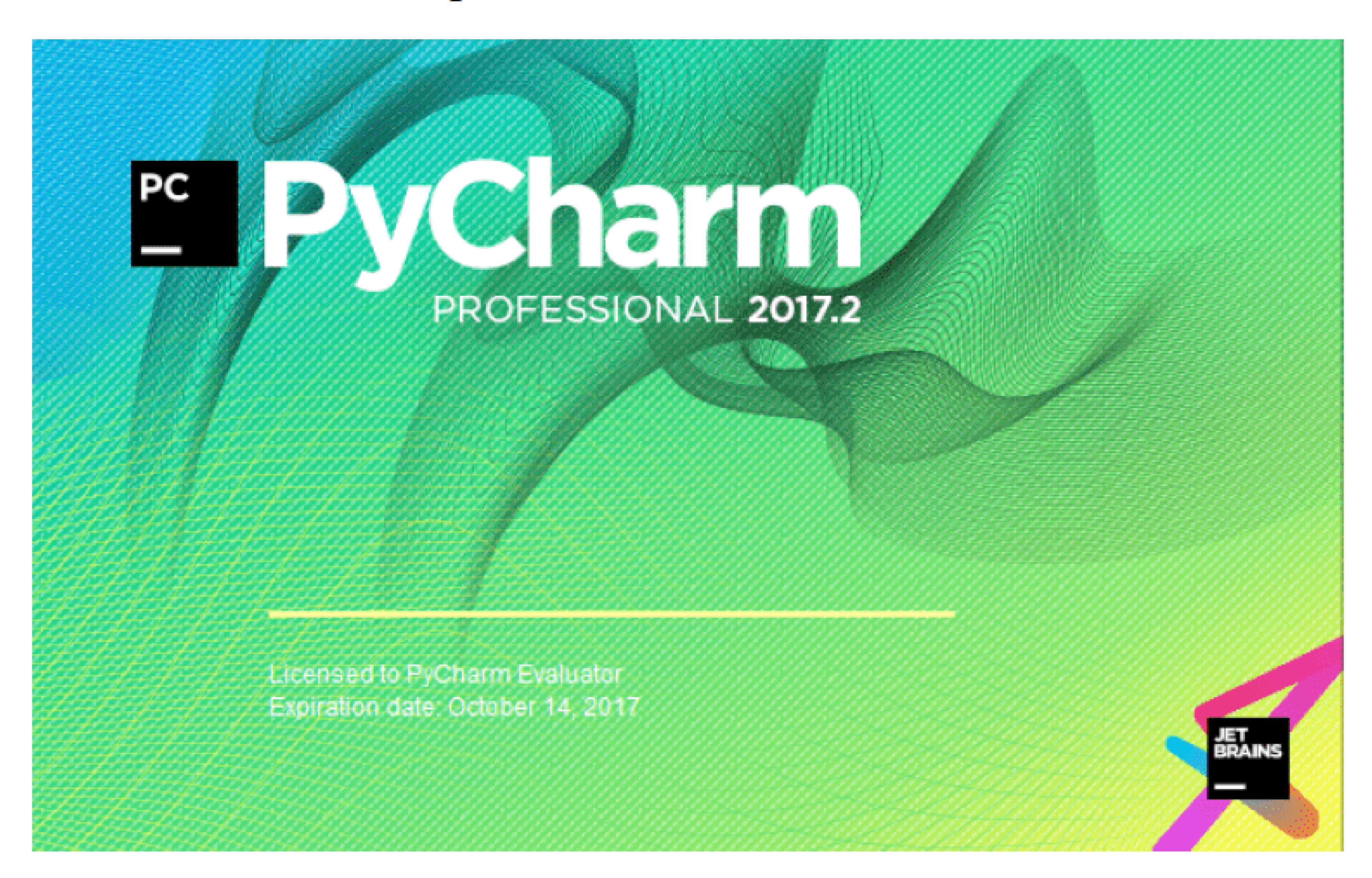


图 1.22 启动界面

（5）启动完成后，进入初始化配置界面，如图 1.23 所示。

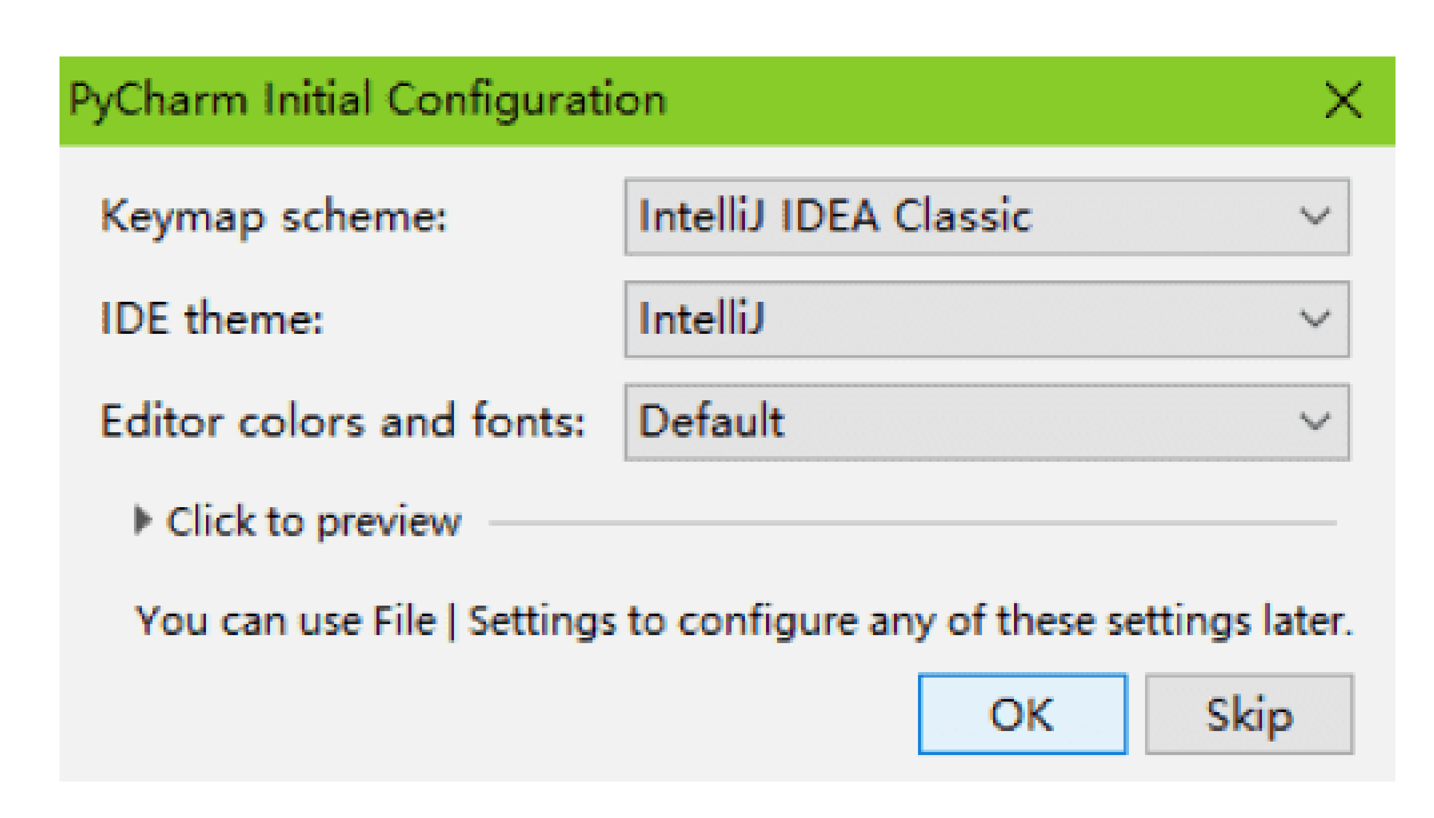


图 1.23 初始化配置

（6）单击图 1.23 中的 OK 按钮，进入创建项目界面，如图 1.24 所示。

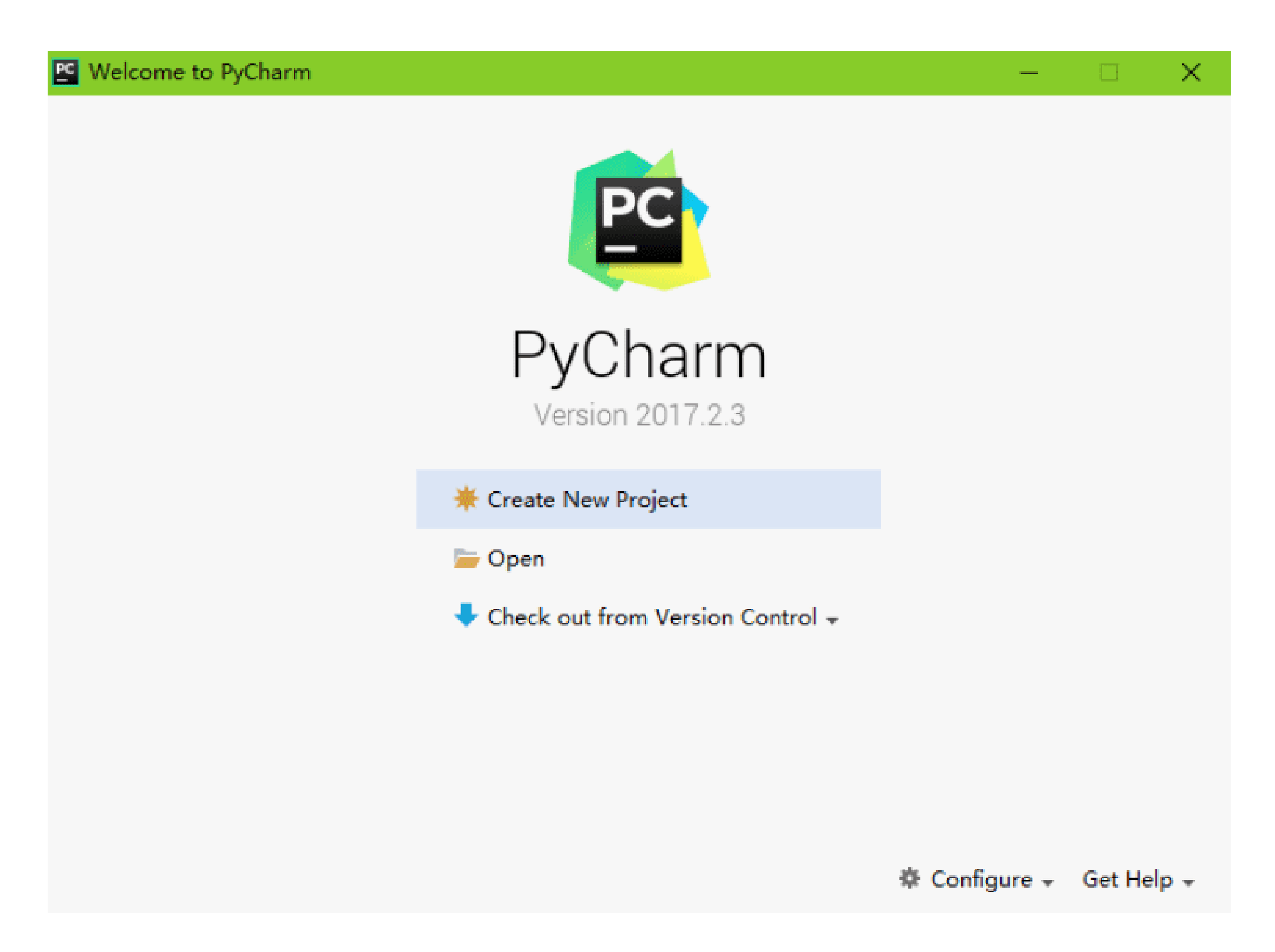


图 1.24　创建项目界面

（7）单击图 1.24 中的 Create New Project 选项，进入项目设置界面，如图 1.25 所示。

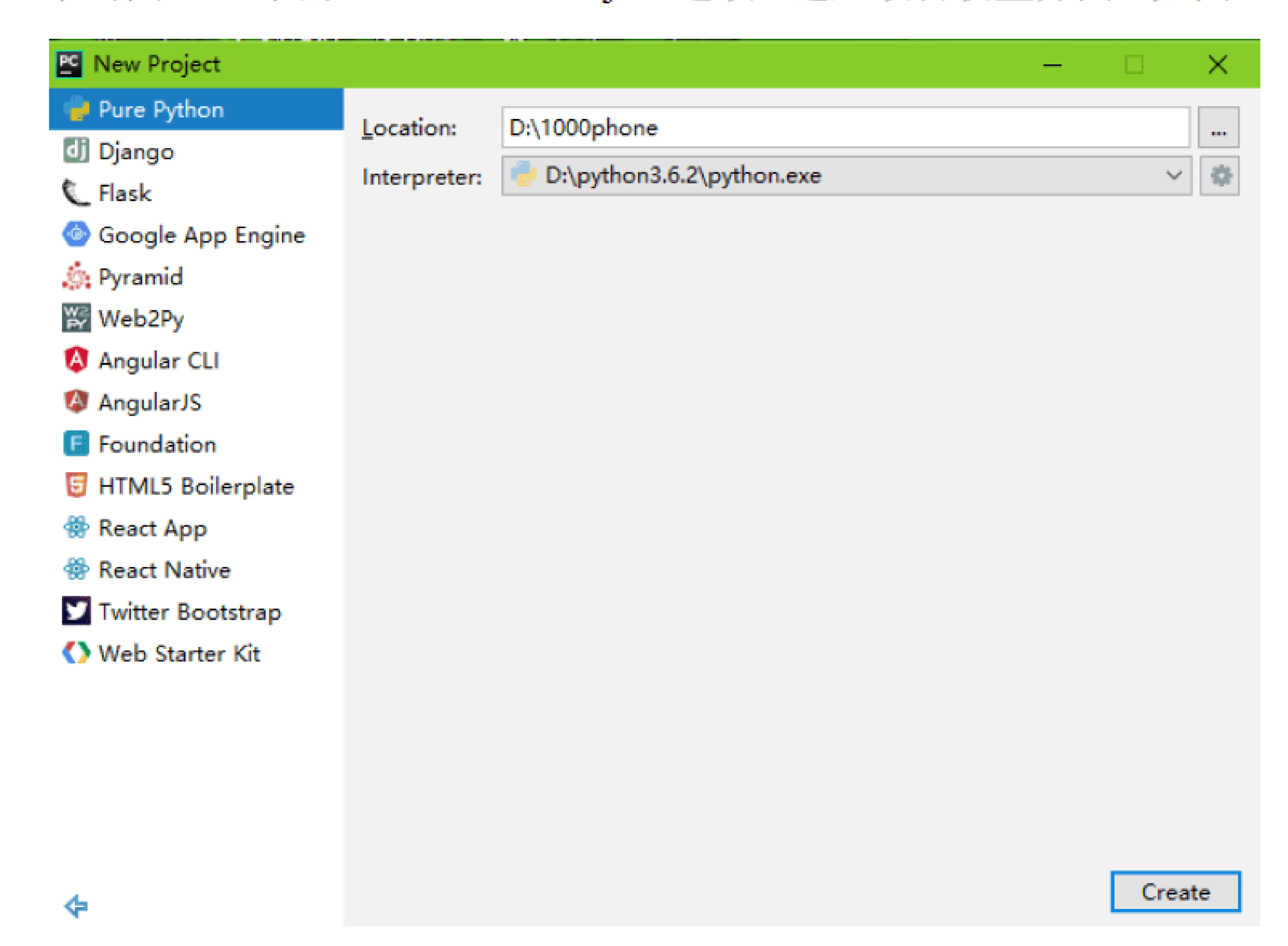


图 1.25　项目设置界面

（8）单击图 1.25 中的 Create 按钮，进入项目开发界面，如图 1.26 所示。

（9）右击图 1.26 中的项目名称，在弹出的快捷菜单中选择 New→Python File 菜单项，如图 1.27 所示。

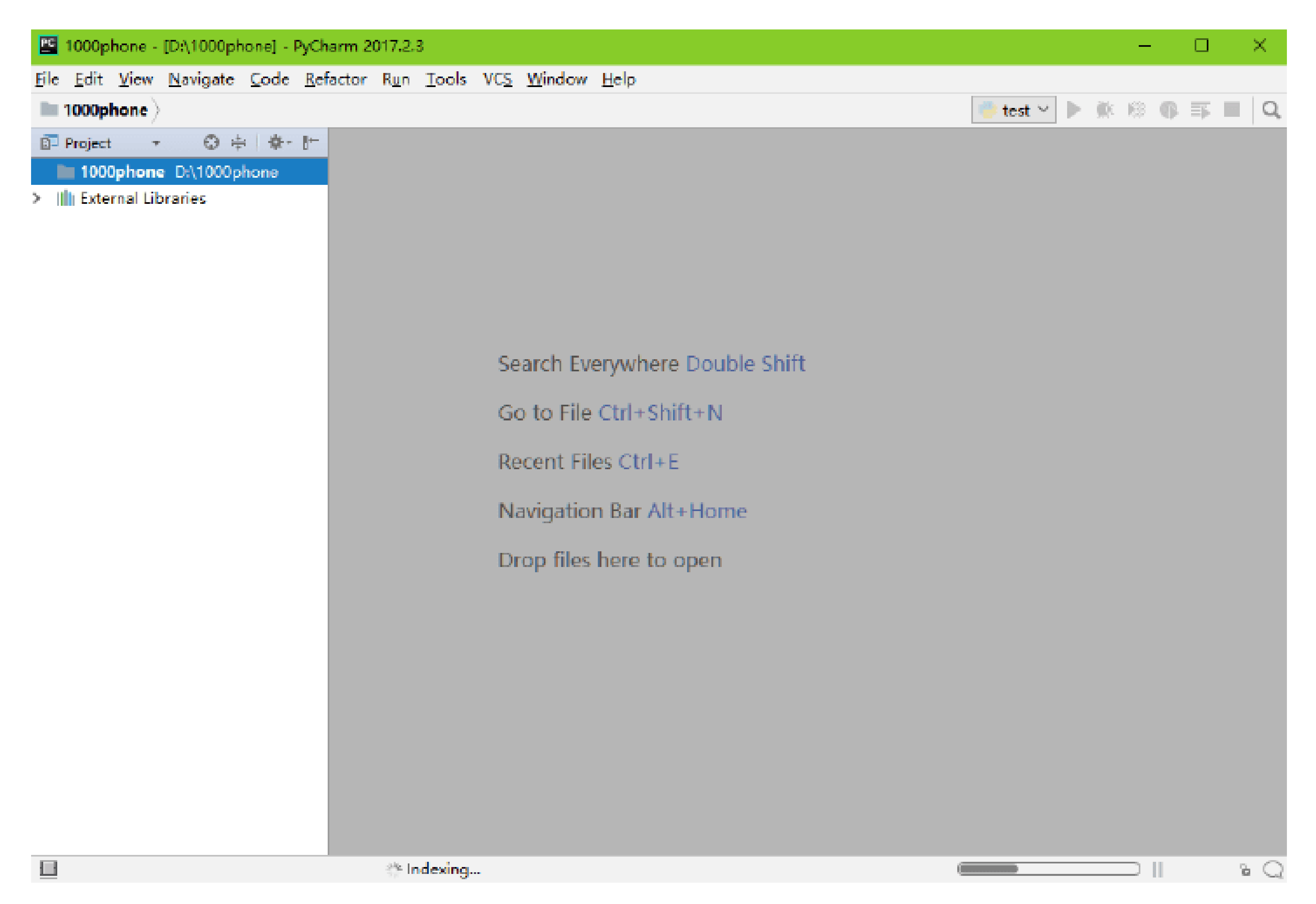


图 1.26　项目开发界面

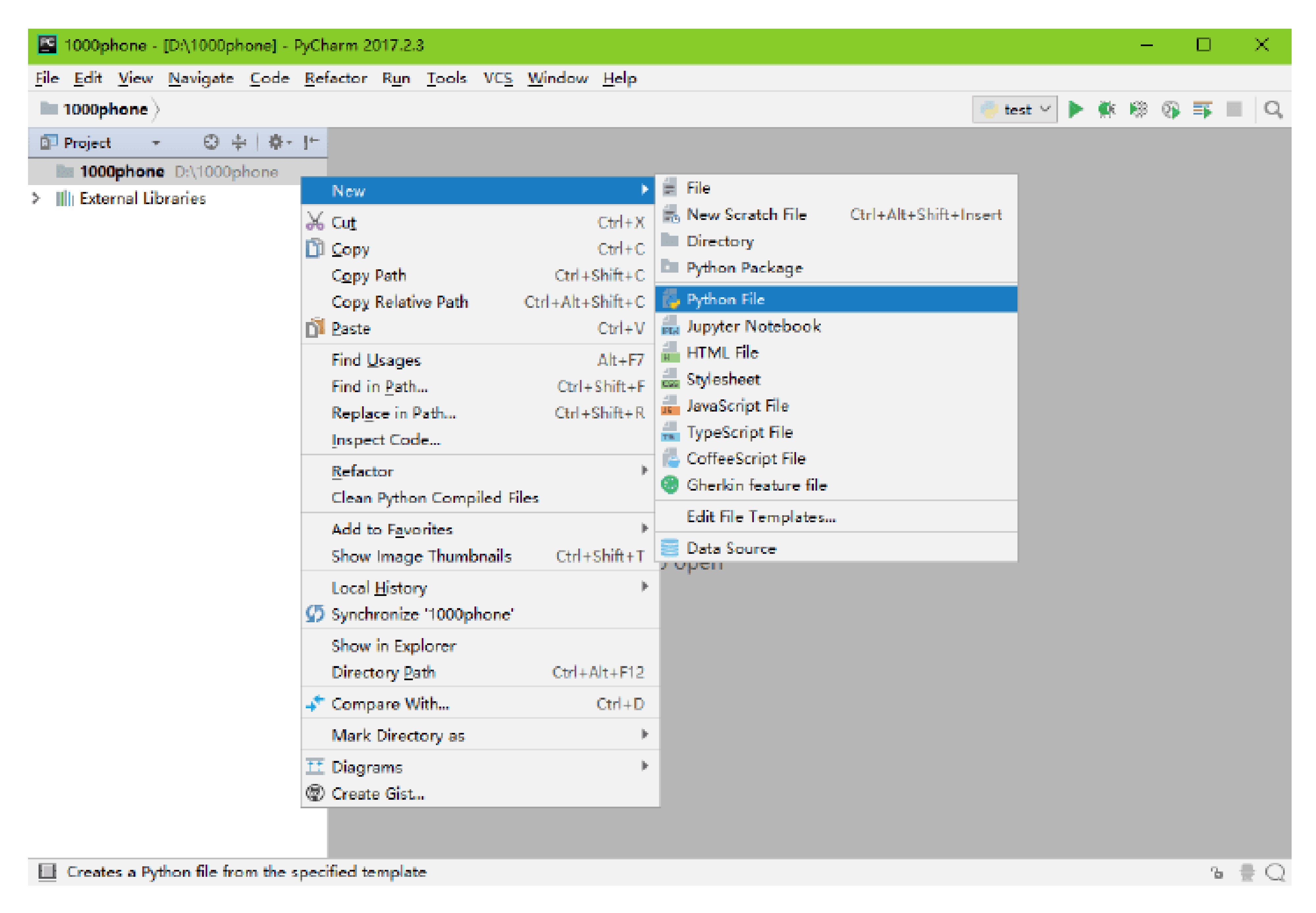


图 1.27　创建新文件

（10）出现填写文件名界面，如图 1.28 所示。

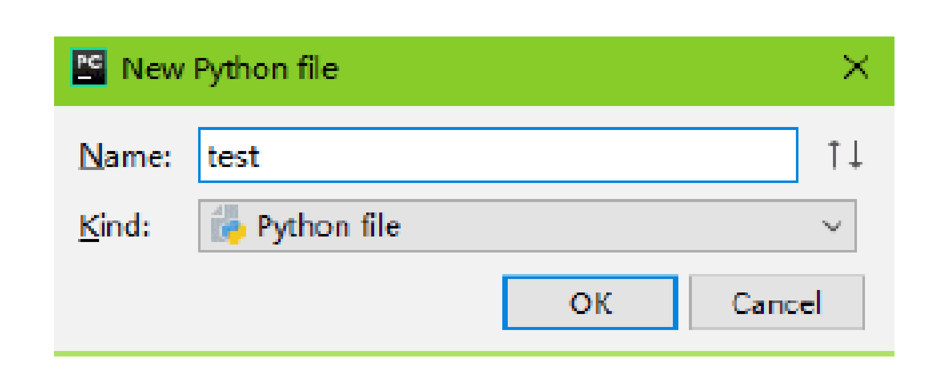


图 1.28　填写文件名界面

（11）在图 1.28 中输入文件名“test”（或“test.py”，默认创建.py 文件）并单击 OK 按钮，则文件创建完成，如图 1.29 所示。

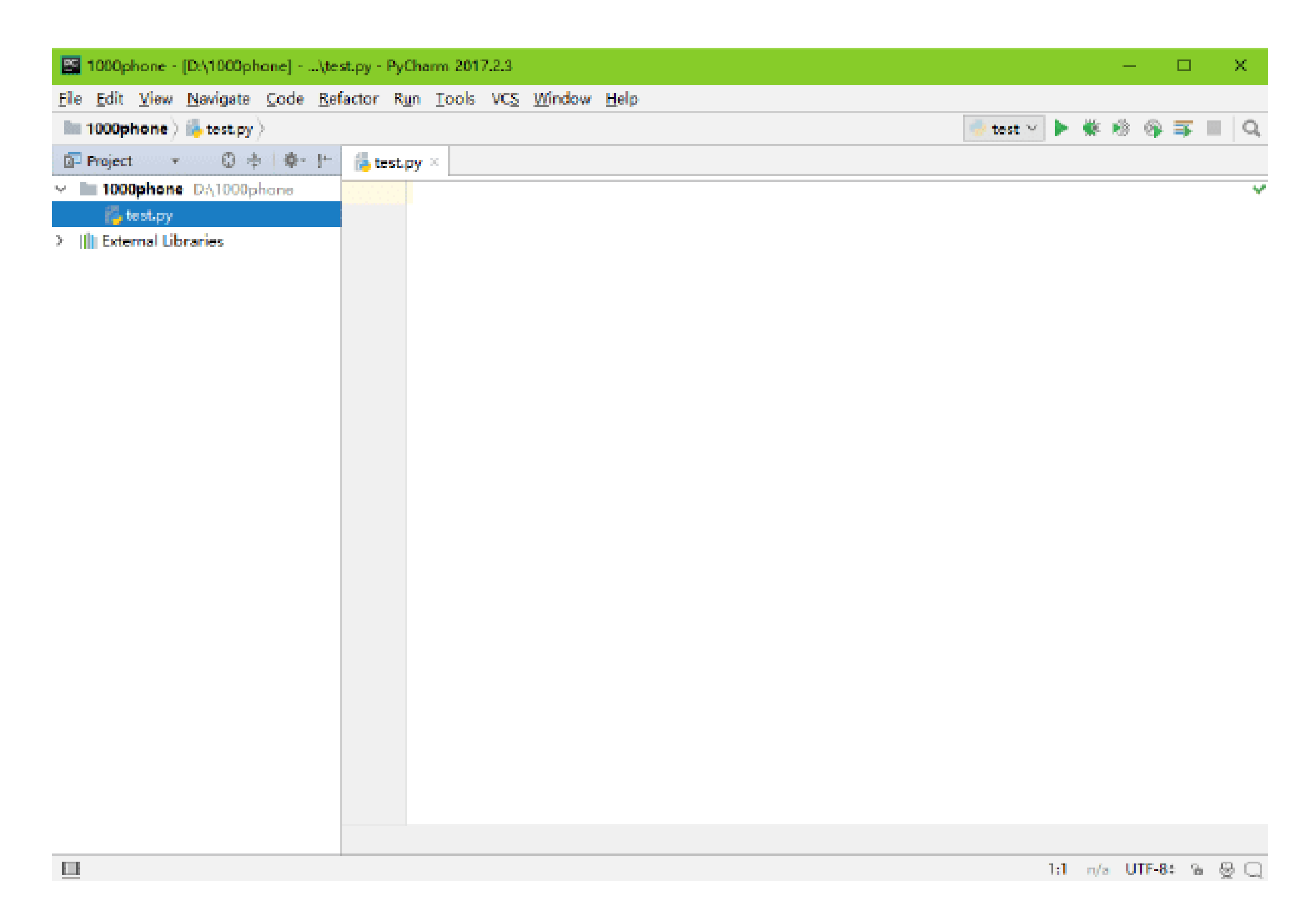


图 1.29　文件创建完成界面

（12）在图 1.29 中，在 test.py 文件编辑区写入如图 1.30 所示的代码。

```
print("Hello world!")
```

图 1.30　编辑代码

（13）右击图 1.30 中的 test.py 文件，在弹出的快捷菜单中选择 Run 'test'选项，如图 1.31 所示。

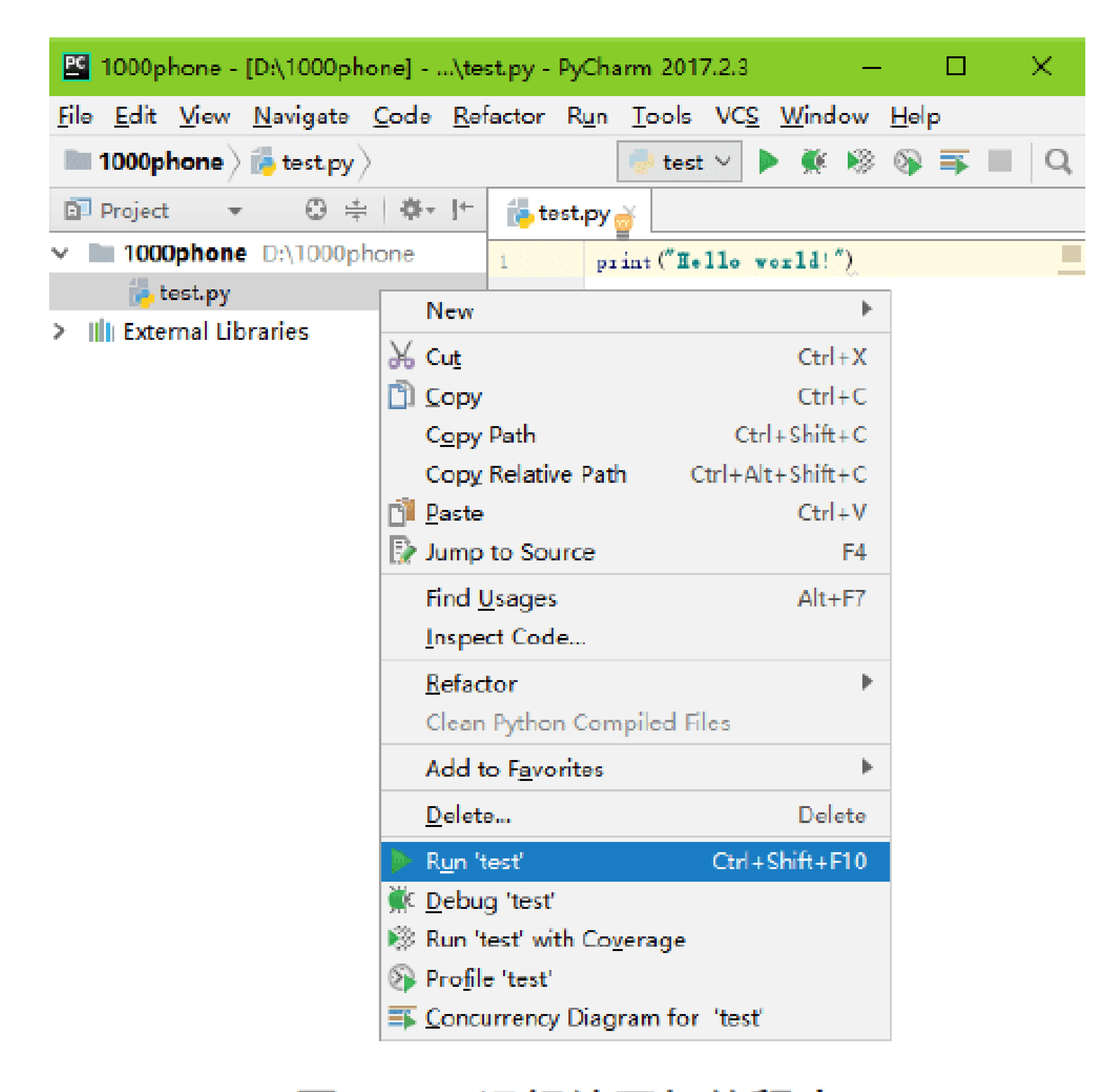


图 1.31　运行编写好的程序

（14）程序运行完后，在下方窗口中可以看到输出结果，如图 1.32 所示。

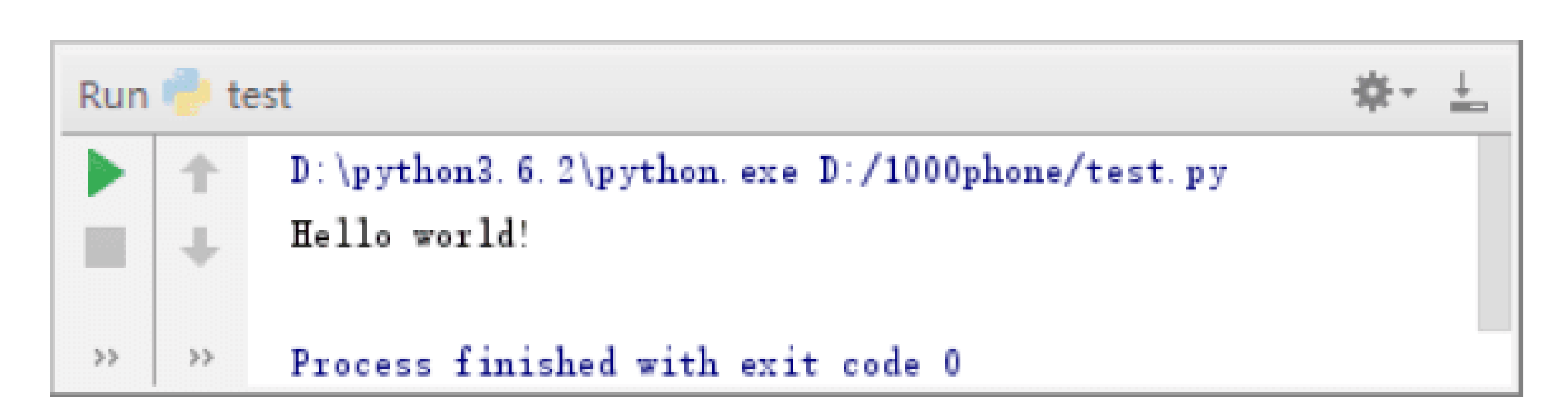


图 1.32 运行结果

以上是使用 PyCharm 实现的字符串打印功能，不管学习哪门语言，当第一个 Hello world 程序成功运行起来的时候，就代表着已经迈进了一小步。

1.4 本章小结

通过本章的学习，相信大家已经对 Python 语言的发展与特性有了初步的认识，应重点掌握 Python 开发环境的搭建，并能编写出一个简单的 Python 程序，为后面学习 Python 开发做好准备。

1.5 习 题

1．填空题

（1）Python 是一种面向________的语言。
（2）Python 3.x 版本的默认编码是________。
（3）Python 程序的默认扩展名是________。
（4）退出 Python 解释器可以输入________。
（5）输出一个字符串可以使用________。

2．选择题

（1）Python 可以在 Windows、Mac 平台运行，体现出 Python 的（　　）特性。
　　A．可移植　　B．可扩展　　C．简单　　D．面向对象
（2）下列不属于 Python 语言特征的选项是（　　）。
　　A．简单易学　　B．免费开源　　C．编译性　　D．面向对象
（3）下列属于 Python 集成开发环境的是（　　）。
　　A．Python　　B．Py　　C．XAMPP　　D．PyCharm
（4）下列属于 Python 应用领域的是（　　）。
　　A．操作系统管理　　B．科学计算

C．Web 应用　　　　　　　　　D．服务器运维的自动化脚本

（5）下列属于 PyCharm 优势的是（　　）。

A．语法高亮　　B．代码跳转　　C．智能提示　　D．版本控制

3．思考题



（1）简述 Python 语言的特性。

（2）简述 Python 3.x 与 Python 2.x 的区别（列出两点即可）。

4．编程题

使用 PyCharm 编写程序，输出“众里寻他千百度，锋自苦寒磨砺出”。

第 2 章 chapter 2

编程基础

本章学习目标

- 掌握 Python 基本语法。
- 掌握变量与数据类型。
- 掌握运算符。

在日常生活中，想要盖一栋房子，那么首先需要知道盖房都需要哪些材料，以及如何将它们组合使用。同样，要使用 Python 开发出一款软件，就必须熟练掌握 Python 的基础知识。

2.1 基本语法

2.1.1 注释

注释即对程序代码的解释，在写程序时需适当使用注释，以方便自己和他人理解程序各部分的作用。在执行时，它会被 Python 解释器忽略，因此不会影响程序的执行。Python 支持单行注释与多行注释，具体如下所示。

1. 单行注释

该注释是以“#”开始，到该行末尾结束，具体示例如下：

```
# 输出千锋教育
print("千锋教育")
```

2. 多行注释

该注释以 3 个引号作为开始和结束符号，其中 3 个引号可以是 3 个单引号或 3 个双引号，具体示例如下：

```
'''
多行注释
输出千锋教育
'''
```

```
"""
多行注释
输出千锋教育
"""
print("千锋教育")
```

2.1.2 标识符与关键字

现实世界中每种事物都有自己的名称，从而与其他事物区分开。例如，生活中每种交通工具都有一个用来标识的名称，如图 2.1 所示。



图 2.1 生活中的标识符

在 Python 语言中，同样也需要对程序中的各个元素命名，以便区分，这种用来标识变量、函数、类等元素的符号称为标识符。

Python 语言规定，标识符由字母、数字和下画线组成，并且是只能以字母或下画线开头的字符集合。在使用标识符时应注意以下几点：

- 命名时应遵循见名知义的原则。
- 系统已用的关键字不得用作标识符。
- 下画线对解释器有特殊的意义，建议避免使用下画线开头的标识符（后续章节进行说明）。
- 标识符区分大小写。

关键字是系统已经定义过的标识符，它在程序中已有了特定的含义，如 if、class 等，因此不能再使用关键字作为其他名称的标识符，表 2.1 列出了 Python 中常用的关键字。

表 2.1 常用的关键字

False	None	True	and	as	assert
break	class	continue	def	del	elif
else	except	finally	for	from	global
if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try
while	with	yield			

Python 的标准库提供了一个 keyword 模块，可以输出当前 Python 版本的所有关键字，具体示例如下：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue'
, 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global',
 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
 'raise', 'return', 'try', 'while', 'with', 'yield']
```

2.1.3 语句换行

Python 中一般是一条语句占用一行，但有时一条语句太长，就需要换行，具体示例如下：

```
print("千锋教育隶属于北京千锋互联科技有限公司，\
一直秉承用良心做教育的理念，致力于打造 IT 教育全产业链人才服务平台。")
print(["千锋教育", "扣丁学堂",
"好程序员特训营"])
```

运行结果如图 2.2 所示。

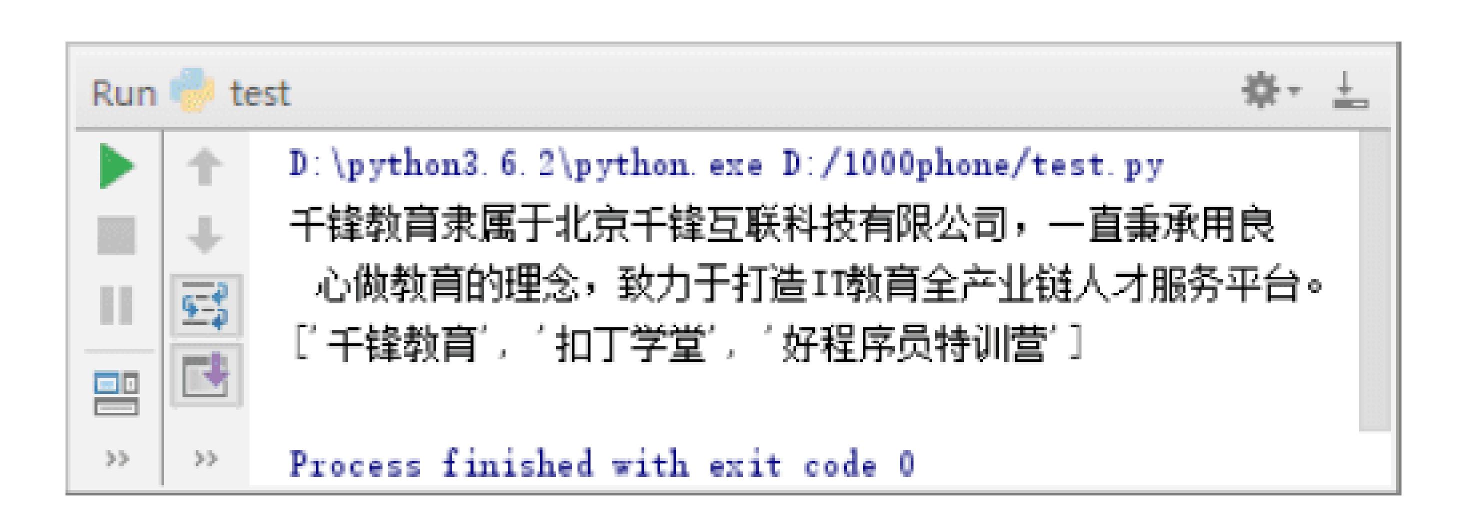


图 2.2 运行结果（一）

示例中，第 1 行 print()中字符串太长，分开两行编写，在首行末尾添加续行符“\”来实现，但在 []、{}中分行时，可以不使用反斜杠，如示例中的第 3 行和第 4 行。

2.1.4 缩进

Python 语言的简洁体现在使用缩进来表示代码块，而不像 C++或 Java 中使用{}，具体示例如下：

```
if True:
    print("如果为真，输出：")
    print("True")
else:
    print("否则，输出：")
    print("False")
```

示例中，if 后的条件为真，执行第 2 行和第 3 行，它们使用相同的缩进来表示一个代码块。此处需要注意，缩进的空格数是可变的，但同一个代码块中的语句必须包含相同的缩进空格。具体示例如下：

```
if True:
        print("如果为真，输出：")
        print("True")
else:
    print("否则，输出：")
        print("False") # 缩进不一致,引发错误
```

示例中，第 5 行与第 6 行缩进不一致，会引发错误。

运行结果如图 2.3 所示。

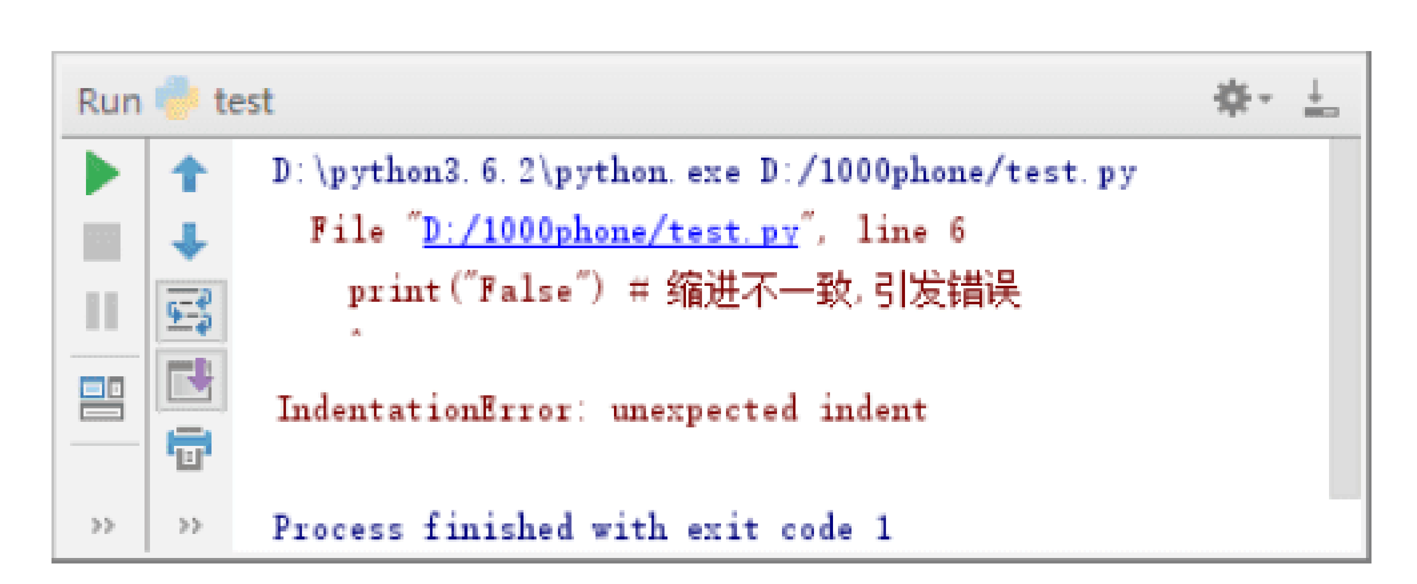


图 2.3　缩进不一致引发错误

在 PyCharm 中，缩进是自动添加的。在其他文本编辑器中使用缩进，推荐大家使用 4 个空格宽度作为缩进，尽量不要使用制表符作为缩进，因为不同的文本编辑器中制表符代表的空白宽度可能不相同。

2.2　变量与数据类型

2.2.1　变量

变量是编程中最基本的单元，它会暂时引用用户需要存储的数据，例如小千的年龄是 18，就可以使用变量来引用 18，如图 2.4 所示。

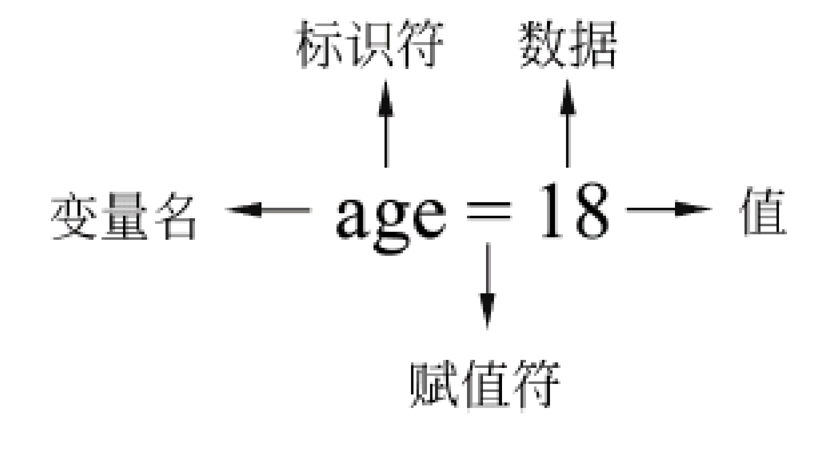


图 2.4　变量

在图 2.4 中，变量名 age 是一个标识符，通过赋值符（=）将数据 18 与变量名 age 建立关系，这样 age 就代表 18，此时可以通过 print()查看 age 的值，具体示例如下：

```
age = 18
print(age)
```

如果想将小千的年龄修改为 20 并输出，则可以使用以下语句：

```
age = 20
print(age)
```

2.2.2 数据类型

在计算机中，操作的对象是数据，那么大家来思考一下，如何选择合适的容器来存放数据才不至于浪费空间？先来看一个生活中的例子，某公司要快递一本书，文件袋和纸箱都可以装载，但是，如果使用纸箱装一本书，显然有点大材小用，浪费纸箱的空间，如图 2.5 所示。



图 2.5　用纸箱或文件袋快递一本书

同理，为了更充分地利用内存空间，可以为不同的数据指定不同的数据类型。Python 的数据类型如图 2.6 所示。

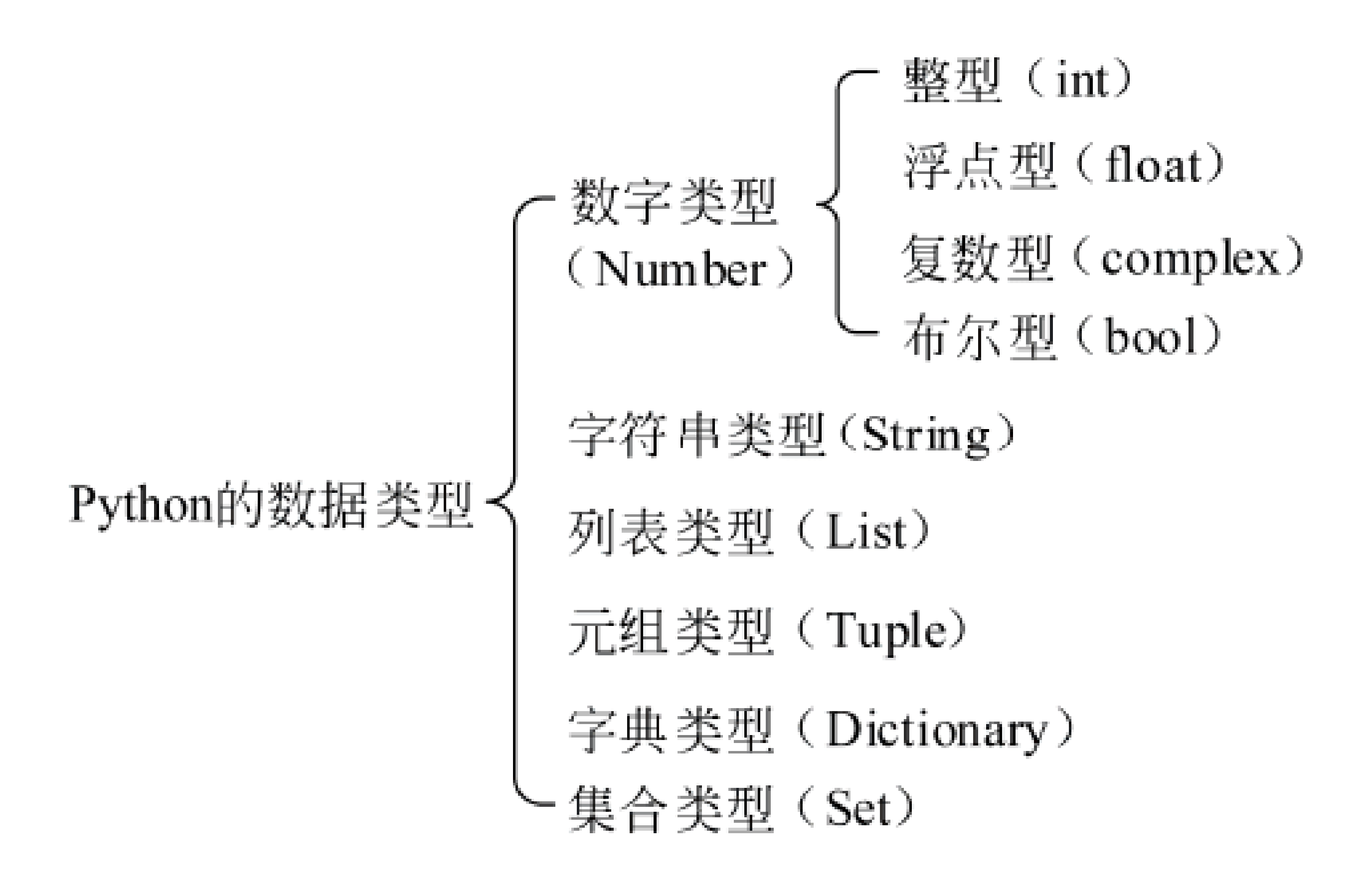


图 2.6　Python 的数据类型

在图 2.6 中，Python 的数据类型分为数字类型（int、float、complex、bool）、字符串类型、列表类型、元组类型、字典类型和集合类型。

1. 整型

整型表示存储的数据是整数，例如 1、–1 等。在计算机语言中，整型数据可以用二进制、八进制、十进制或十六进制形式并在前面加上“+”或“–”表示。如果用二进制表示，那么数字前必须加上 0b 或 0B；如果用八进制表示，那么数字前必须加上 0o 或 0O；如果用十六进制表示，那么数字前必须加上 0x 或 0X。具体示例如下：

```
a = 0b1010   # 二进制数,等价于十进制数 10
b = -0b1010  # 二进制数,等价于十进制数-10
c = 10       # 十进制数 10
d = -10      # 十进制数-10
e = -0o12    # 八进制数,等价于十进制数-10
f = -0XA     # 十六进制数,等价于十进制数-10
```

八进制数是由 0～7 的数字序列组成的，每逢 8 进 1 位；十六进制数是由 0～9 的数字和 A～F 的字母组成序列，每逢 16 进 1 位。此处需要注意，整型数值有最大取值范围，其范围与具体平台的位数有关。

2. 浮点型

浮点型表示存储的数据是实数，如 3.145。在 Python 中，浮点型数据默认有两种书写格式，具体示例如下：

```
f1 = 0.314      #  标准格式
f2 = 31.4e-2    #  科学记数法格式,等价于 0.314
f3 = 31.4E2     #  科学记数法格式,等价于 3140.0
```

在科学记数法格式中，E 或 e 代表基数是 10，其后的数字代表指数，31.4e–2 表示 31.4×10^{-2}，31.4E2 表示 31.4×10^{2}。

3. 复数型

复数型用于表示数学中的复数，如 1+2j、1–2j、–1–2j 等，这种类型在科学计算中经常使用，其语法格式如下：

```
a = 3 + 1j
print(a.real) # 打印实部
print(a.imag) # 打印虚部
```

此处需要注意它的写法与数学中写法的区别，当虚部为 1j 或–1j 时，在数学中，可以省略 1，但在 Python 程序中，1 是不可以省略的。

4. 布尔型

布尔型是一种比较特殊的整型，它只有 True 和 False 两种值，分别对应 1 和 0。它

主要用来比较和判断，所得结果叫作布尔值。具体示例如下：

```
3 == 3  # 结果为True
3 == 4  # 结果为False
```

此外，每一个 Python 对象都有一个布尔值，从而可以进行条件测试，下面对象的布尔值都为 False：

```
None
False(布尔型)
0(整型 0)
0.0(浮点型 0)
0.0 + 0.0j(复数型 0)
""(空字符串)
[](空列表)
()(空元组)
{}(空字典)
```

除上述对象外，其他对象的布尔值都为 True。

2.2.3 检测数据类型

在 Python 中，数据类型是由存储的数据决定的。为了检测变量所引用的数据是否符合期望的数据类型，Python 中内置了检测数据类型的函数 type()。它可以对不同类型的数据进行检测，具体如下所示：

```
a = 10
print(type(a)) # <class 'int'>
b = 1.0
print(type(b)) # <class 'float'>
c = 1.0 + 1j
print(type(c)) # <class 'complex'>
```

示例中，使用 type()函数分别检测 a、b、c 所引用数据的类型。

除此之外，还可以使用函数 isinstance()判断数据是否属于某个类型，具体示例如下：

```
a = 10
print(isinstance(a, int))       # 输出True
print(isinstance(a, float))     # 输出False
```

2.2.4 数据类型转换

数据类型转换是指数据从一种类型转换为另一种类型，转换时，只需要将目标数据类型名作为函数名即可，如表 2.2 所示。

表 2.2 数据类型转换函数

函　　数	说　　明
int(x [,base = 10])	将一个数字或 base（代表进制）类型的字符串转换成整数
float(x)	将 x 转换为一个浮点数
complex(real [,imag])	创建一个复数

表 2.2 中列出的是数字类型之间的转换，其他类型之间也可以转换，如数字类型转换为字符串型，这些知识将在后面章节中讲解。

接下来演示数字类型之间的转换，如例 2-1 所示。

例 2-1 数字类型之间的转换。

```
1   a = 2.0
2   print(type(a))
3   print(int(a))             # 将浮点型转为整型
4   print(type(int(a)))
5   b = 2
6   print(type(b))
7   print(float(b))           # 将整型转为浮点型
8   print(type(float(b)))
9   c = complex(2.3, 1.2)     # 创建一个复数
10  print(c)
11  print(type(c))
```

运行结果如图 2.7 所示。

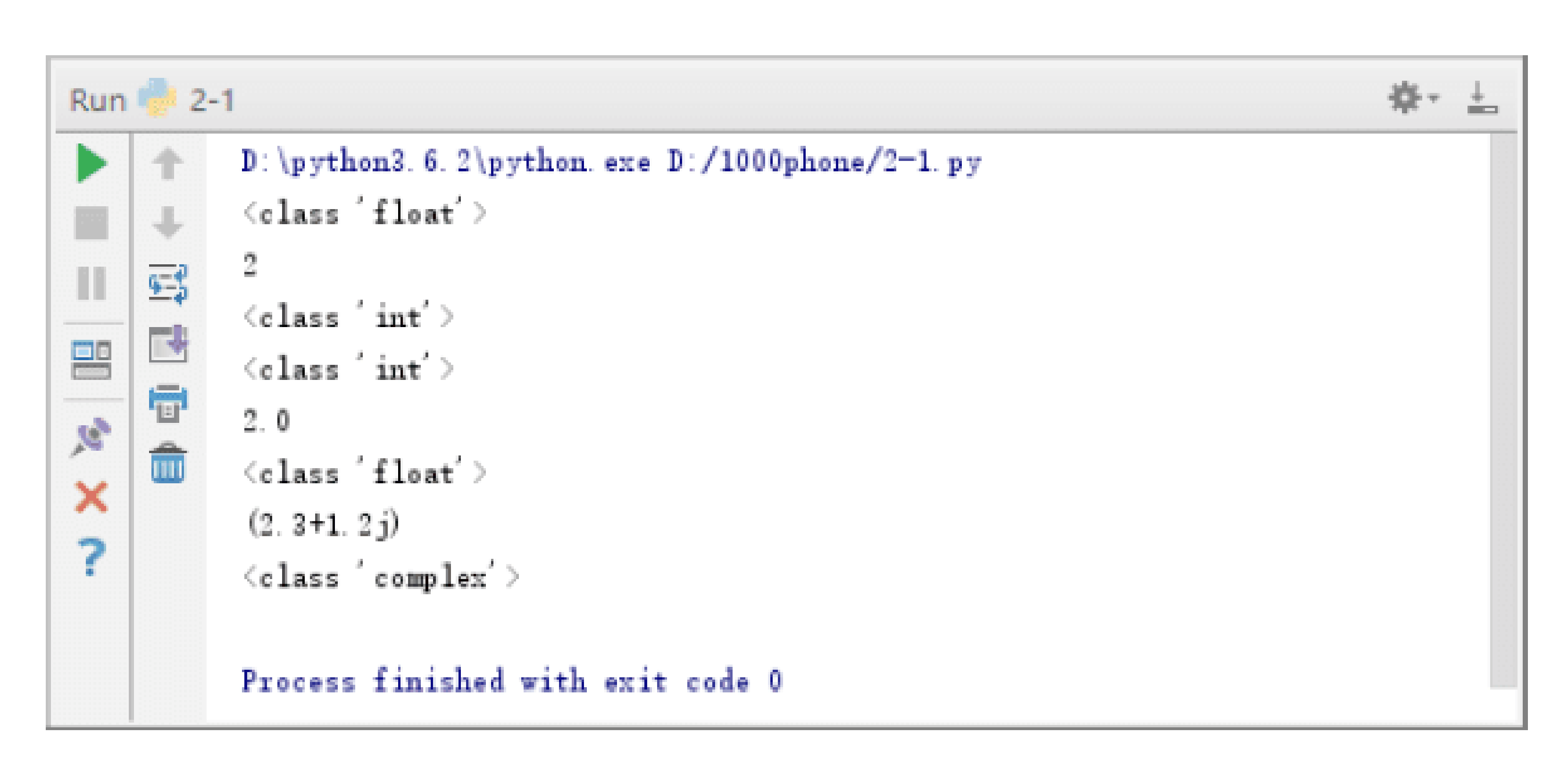


图 2.7 例 2-1 运行结果

在例 2-1 中，第 3 行使用 int()函数将浮点型数据转为整型数据并通过 print()函数输出，第 7 行使用 float()函数将整型数据转为浮点型数据并通过 print()函数输出，第 9 行使用 complex()函数创建复数 2.3+1.2j，可以认为将浮点数转为复数。

2.3 运　算　符

运算符是用来对变量或数据进行操作的符号，也称作操作符，操作的数据称为操作

数。运算符根据其功能可分为算术运算符、赋值运算符、比较运算符、逻辑运算符等。

2.3.1 算术运算符

算术运算符用来处理简单的算术运算，包括加、减、乘、除、取余等，具体如表 2.3 所示。

表 2.3 算术运算符

运 算 符	说 明	示 例	结 果
+	加	5 + 2	7
–	减	5 – 2	3
*	乘	5 * 2	10
/	除	5 / 2	2.5
%	取余	5 % 2	1
**	幂	5**2	25
//	取整	5 // 2	2

在表 2.3 中，注意除法与取整的区别。接下来演示两者的区别，如例 2-2 所示。

例 2-2 除法与取整的区别。

```
print(10/2)      # 除法
print(5/2)
print(5.0/2)
print(5/2.0)
print(10//2)     # 取整
print(5//2)
print(5.0//2)
print(5//2.0)
```

运行结果如图 2.8 所示。

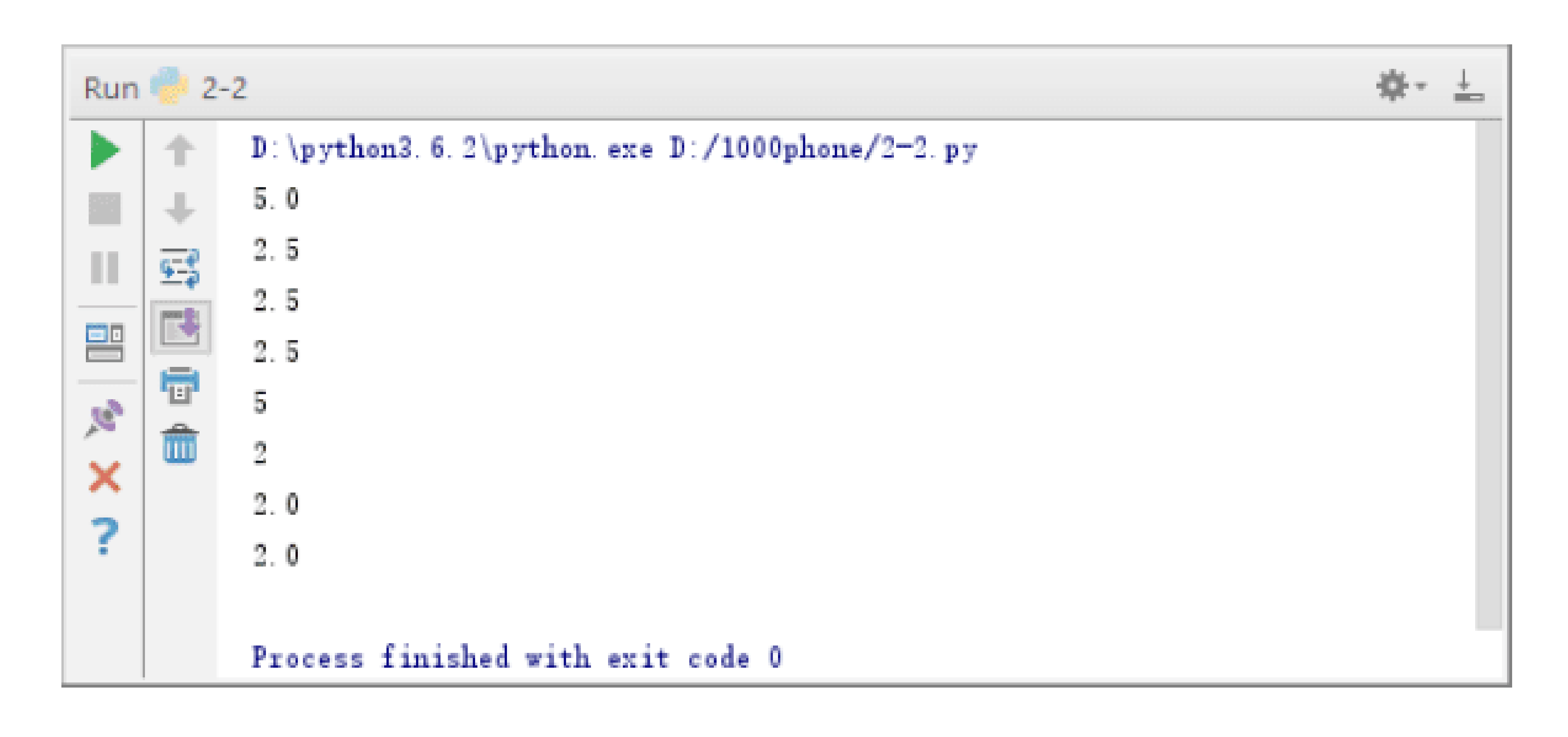


图 2.8 例 2-2 运行结果

在例 2-2 中，从运行结果可看出，进行除法运算的结果始终是浮点数，进行取整运算的结果可能是整数，也可能是浮点数，只要两个操作数中有一个为浮点数，则结果就为浮点数。

2.3.2 赋值运算符

在前面章节的学习中，程序中已多次使用赋值运算符，它的作用就是将变量或表达式的值赋给某一个变量，具体示例如下：

```
a = 13
b = a + 1   # b为14
```

如果需要为多个变量赋相同的值，可以简写为如下形式：

```
a = b = 13
```

上述语句等价于如下语句：

```
a = 13
b = 13
```

如果需要为多个变量赋不同的值，可以简写为如下形式：

```
a, b, c, d = 13, 3.14, 1 + 2j, True
```

输出 a、b、c、d 的值时，可以使用如下语句：

```
print(a, b, c, d)
```

除此之外，还有几种特殊的赋值运算符，如表 2.4 所示。

表 2.4 复合赋值运算符

运　算　符	说　　明	示　　例
+=	加等于	a += b 等价于 a = a + b
-=	减等于	a -= b 等价于 a = a - b
*=	乘等于	a *= b 等价于 a = a * b
/=	除等于	a /= b 等价于 a = a / b
%=	余等于	a %= b 等价于 a = a % b
**=	幂等于	a **= b 等价于 a = a ** b
//=	整除等于	a //= b 等价于 a = a // b

接下来演示赋值运算符的用法，如例 2-3 所示。

例 2-3 赋值运算符的用法。

```
a, b = 5, 2
a += b
print(a, b)
a -= b
print(a, b)
a *= b
print(a, b)
```

```
a /= b
print(a, b)
a **= b
print(a, b)
a //= b
print(a, b)
```

运行结果如图 2.9 所示。

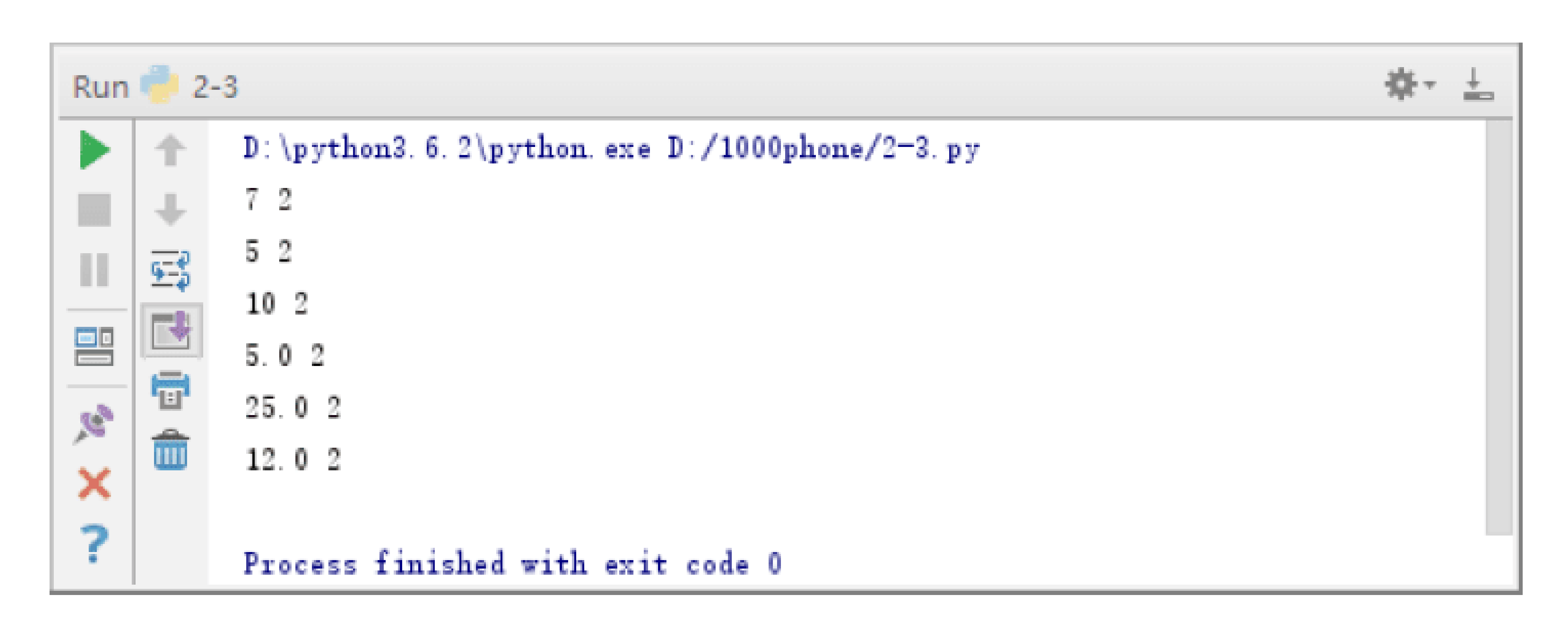


图 2.9　例 2-3 运行结果

在例 2-3 中，使用不同的赋值运算符对 a、b 进行运算，并将运算结果输出。从运行结果可发现，b 的值始终不变。

2.3.3　比较运算符

比较运算符就是对变量或表达式的结果进行比较。如果比较结果为真，则返回 True，否则返回 False。具体如表 2.5 所示。

表 2.5　比较运算符

运　算　符	说　　明	示　　例	结　　果
==	等于	5 == 3	False
!=	不等于	5 != 3	True
>	大于	5 > 3	True
>=	大于或等于	5 >= 3	True
<	小于	5 < 3	False
<=	小于或等于	5 <= 3	False

接下来演示比较运算符的使用，如例 2-4 所示。

例 2-4　比较运算符的使用。

```
print(1 == 1)
print(1 == 2)
print(1 == True)
print(0 == False)
print(1.0 == True)
```

```
print(0.0 == False)
print((0.0 + 0.0j) == False)
```

运行结果如图 2.10 所示。

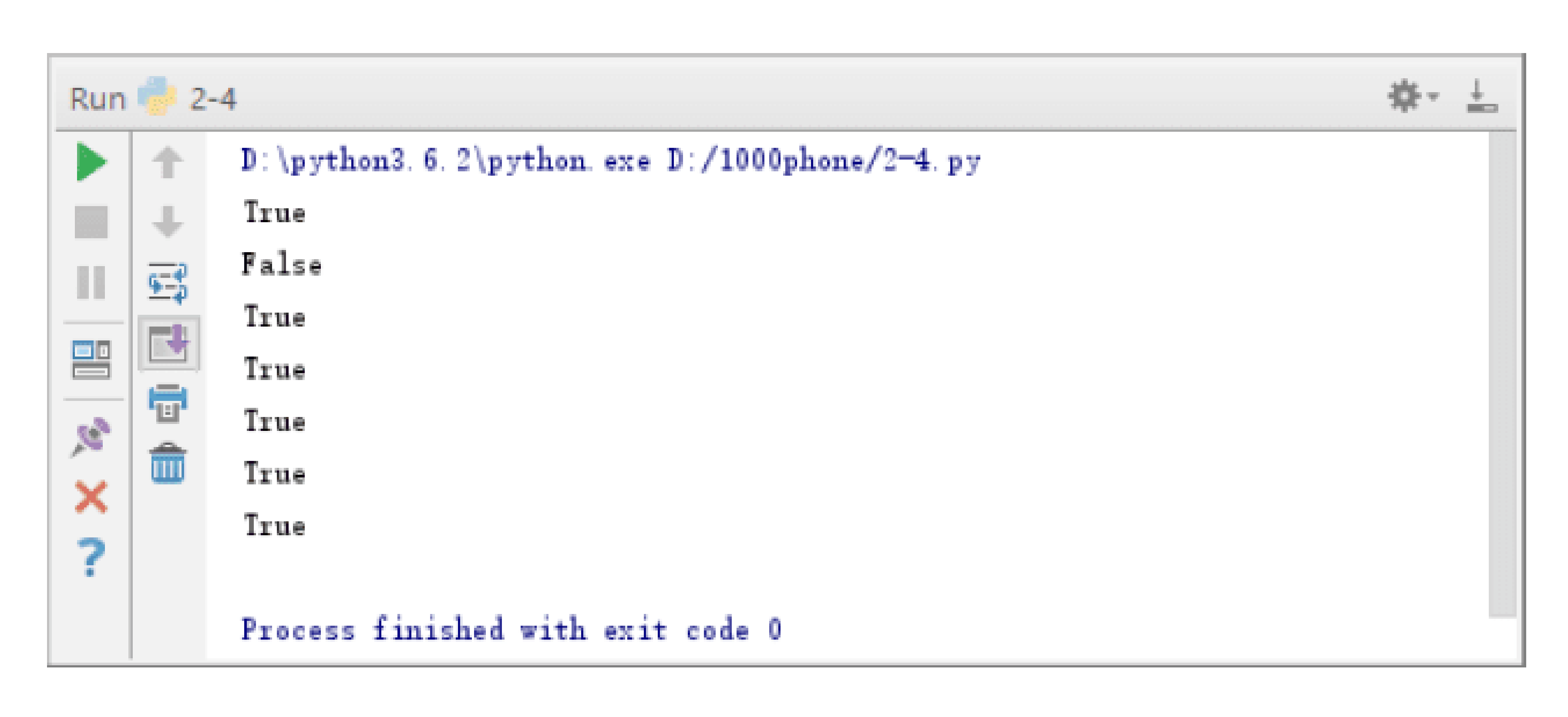


图 2.10　例 2-4 运行结果

在例 2-4 中，注意 1、1.0 与 True 进行等于运算后，结果为 True；0、0.0、0.0 + 0.0j 与 False 进行等于运算后，结果为 True。

2.3.4　逻辑运算符

逻辑运算符用来表示数学中的“与”“或”“非”运算，具体如表 2.6 所示。

表 2.6　逻辑运算符

运　算　符	说　　明	示　　例	结　　果
and	与	a and b	如果 a 的布尔值为 True，返回 b，否则返回 a
or	或	a or b	如果 a 的布尔值为 True，返回 a，否则返回 b
not	非	not a	a 为 False，返回 True；a 为 True，返回 False

在表 2.6 中，a、b 分别为表达式，通常都是使用比较运算符返回的结果作为逻辑运算符的操作数。此外，逻辑运算符也经常出现在条件语句和循环语句中。

接下来演示逻辑运算符的使用，如例 2-5 所示。

例 2-5　逻辑运算符的使用。

```
print(0 and 4)
print(False and 4)
print(1 and 4)
print(1 or 4)
print(True or 4)
print(0 or 4)
print((4 <= 5) and (4 >= 3))
print((4 >= 5) or (4 <= 3))
print(not 1)
```

运行结果如图 2.11 所示。

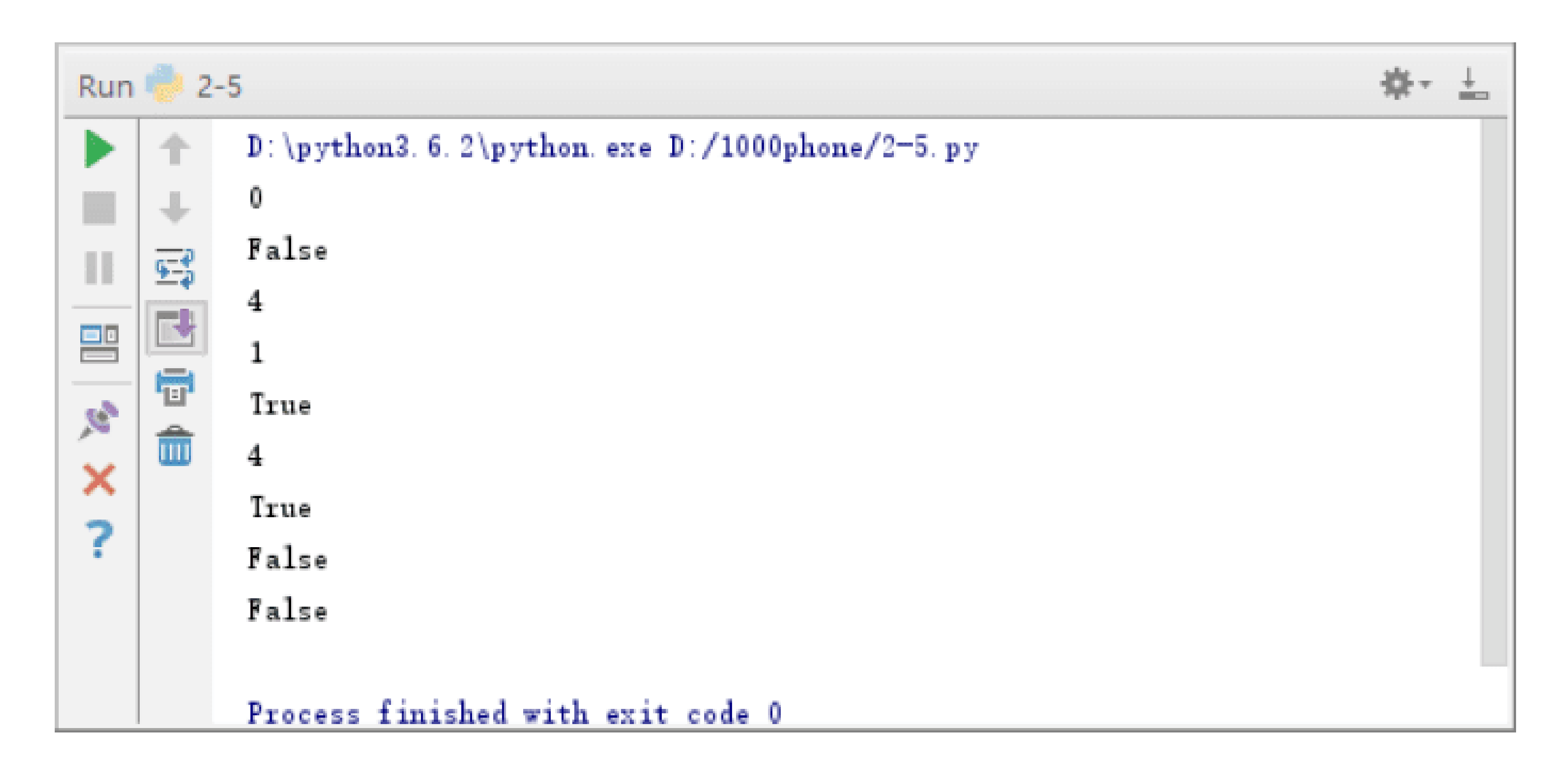


图 2.11　例 2-5 运行结果

在例 2-5 中，程序通过 print()函数输出各个逻辑表达式的值。

2.3.5　位运算符

位运算符是指对二进制位从低位到高位对齐后进行运算，具体如表 2.7 所示。

表 2.7　位运算符

运 算 符	说　　明	示　　例	结　　果
&	按位与	a & b	a 与 b 对应二进制的每一位进行与操作后的结果
|	按位或	a | b	a 与 b 对应二进制的每一位进行或操作后的结果
^	按位异或	a ^ b	a 与 b 对应二进制的每一位进行异或操作后的结果
～	按位取反	～a	a 对应二进制的每一位进行非操作后的结果
<<	向左移位	a << b	将 a 对应二进制的每一位左移 b 位，右边移空的部分补 0
>>	向右移位	a >> b	将 a 对应二进制的每一位右移 b 位，左边移空的部分补 0

虽然运用位运算可以完成一些底层的系统程序设计，但 Python 程序很少涉及计算机底层的技术，因此这里只需要简单了解位运算即可。

接下来演示位运算符的使用，如例 2-6 所示。

例 2-6　位运算符的使用。

```
a, b = 7, 8
print(bin(a))  # 二进制形式 111
print(bin(b))  # 二进制形式 1000
print(bin(a & b))
print(bin(a | b))
print(bin(a ^ b))
print(bin(~a))
print(bin(a << 2))
print(bin(a >> 2))
```

运行结果如图 2.12 所示。

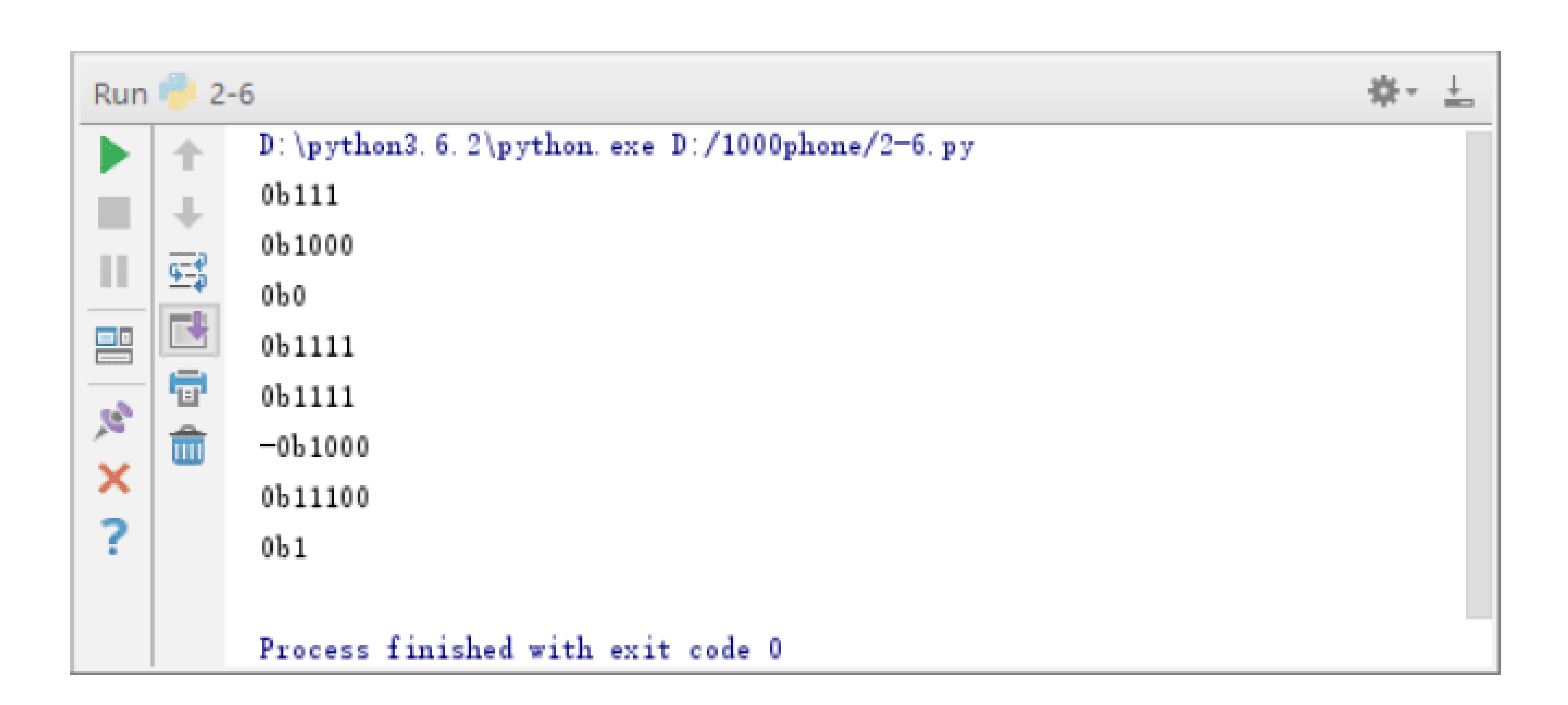


图 2.12　例 2-6 运行结果

在例 2-6 中，程序通过 print()函数输出各个位运算符参与表达式的值。bin()的作用是将数据转换为二进制形式。

2.3.6　成员运算符

成员运算符用于判断指定序列中是否包含某个值，具体如表 2.8 所示。

表 2.8　成员运算符

运　算　符	说　　明
in	如果在指定序列中找到值，则返回 True，否则返回 False
not in	如果在指定序列中找到值，则返回 False，否则返回 True

接下来演示成员运算符的使用，如例 2-7 所示。

例 2-7　成员运算符的使用。

```
A = [1, 2, 3, 4]  # 列表
print(1 in A)
print(0 in A)
print(1 not in A)
print(0 not in A)
```

运行结果如图 2.13 所示。

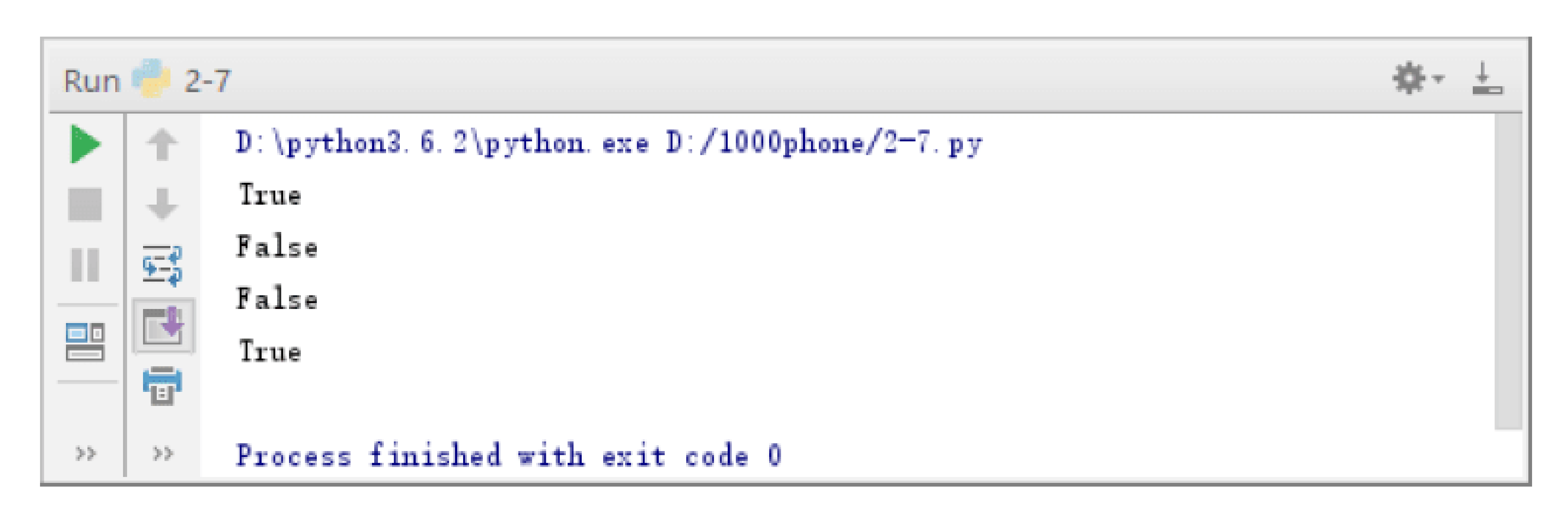


图 2.13　例 2-7 运行结果

在例 2-7 中，程序通过 print()函数输出每个成员运算符参与表达式的值。

2.3.7 身份运算符

身份运算符用于判断两个标识符是否引用同一对象，具体如表 2.9 所示。

表 2.9 身份运算符

运 算 符	说 明
is	如果两个标识符引用同一对象，则返回 True，否则返回 False
is not	如果两个标识符引用同一对象，则返回 False，否则返回 True

接下来演示身份运算符的使用，如例 2-8 所示。

例 2-8 身份运算符的使用。

```
a = b = 10  # a、b 都为 10
print(a is b)
print(a is not b)
b = 20      # b 修改为 20
print(a is b)
print(a is not b)
```

运行结果如图 2.14 所示。

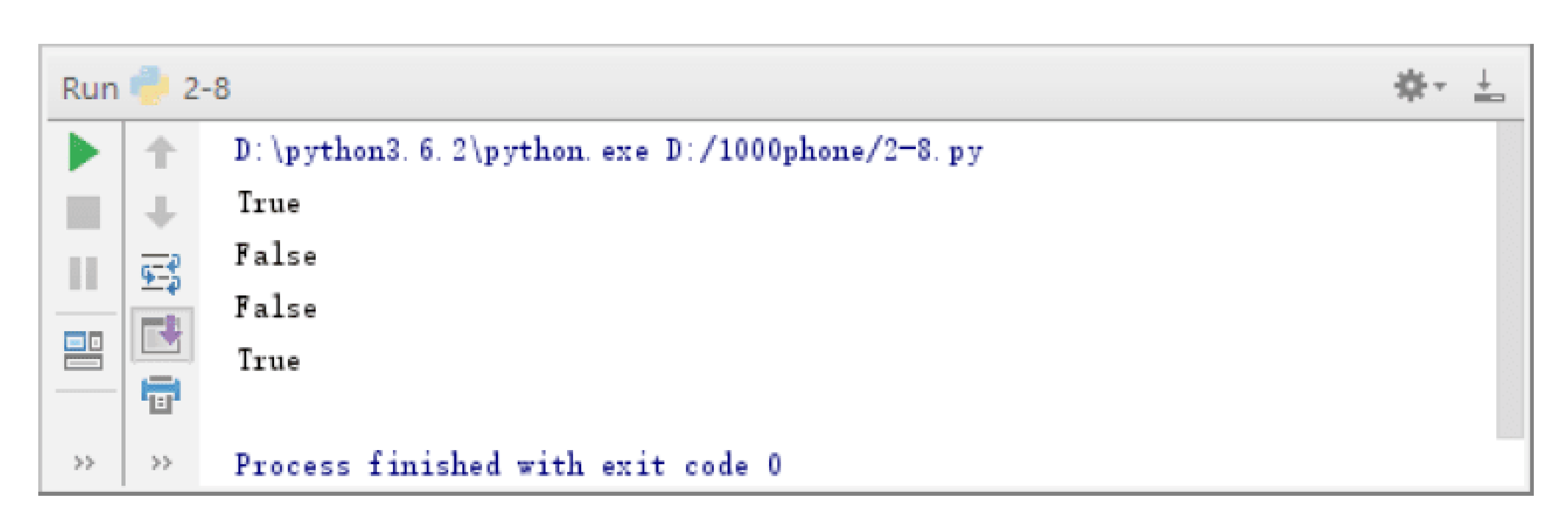


图 2.14 例 2-8 运行结果

在例 2-8 中，程序通过 print()函数输出每个身份运算符参与表达式的值。

2.3.8 运算符的优先级

运算符的优先级是指在多种运算符参与运算的表达式中优先计算哪个运算符，与算术运算中“先乘除，后加减”是一样的。如果运算符的优先级相同，则根据结合方向进行计算。表 2.10 中列出了运算符优先级从高到低的顺序。

表 2.10 运算符优先级

运 算 符	说 明
**	幂
～	按位取反

续表

运　算　符	说　　明
*、/、%、//	乘、除、取余、整除
+、–	加、减
<<、>>	左移、右移
&	按位与
^	按位异或
|	按位或
<=、<、>、>=、==、! =	比较运算符
=、%=、/=、//=、*=、**=、+=、–+	赋值运算符
is、is not	身份运算符
in、not in	成员运算符
not	非运算符
and	与运算符
or	或运算符

Python 会根据表 2.10 中运算符的优先级确定表达式的求值顺序，同时还可以使用小括号“()”来控制运算顺序。小括号内的运算将最先计算，因此在程序开发中，编程者不需要刻意记忆运算符的优先级顺序，而是通过小括号来改变优先级以达到目的。

2.4 小　案　例

从键盘输入一个 3 位整数，计算并输出其百位、十位和个位上的数字，具体实现如例 2-9 所示。

例 2-9　输出 3 位整数的百位、十位和个位上的数字。

```
x = input('请输入一个三位整数：')    # 从键盘输入字符串
x = int(x)                          # 将字符串转换为整数
a = x // 100                        # 获取百位上数字
b = x // 10 % 10                    # 获取十位上数字
c = x % 10                          # 获取个位上数字
print('百位:', a, '十位:', b, '个位:', c)
```

程序运行时，从键盘输入 356，则运行结果如图 2.15 所示。

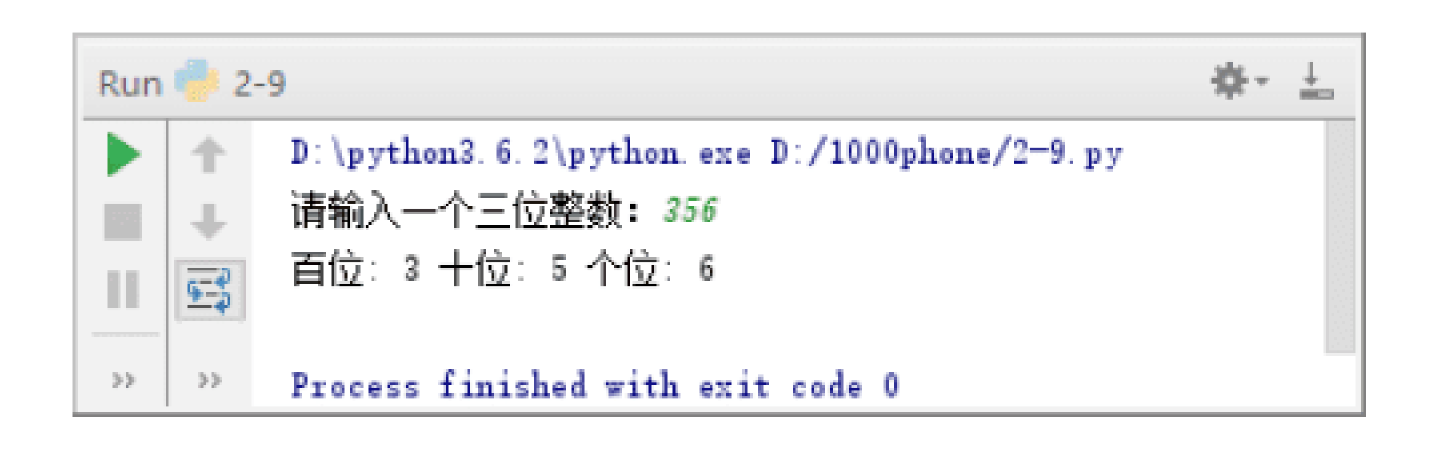


图 2.15　例 2-9 运行结果

在例 2-9 中，通过使用//和%运算符可以获取一个 3 位整数百位、十位和个位上的数

字。在后面学习 map()函数后，还可以使用以下方法解决，具体如例 2-10 所示。

例 2-10 使用 map()函数获取一个 3 位整数百位、十位和个位上的数字。

```
x = input('请输入一个三位整数：')
a, b, c = map(int, x)
print('百位:', a, '十位:', b, '个位:', c)
```

程序运行时，从键盘输入 356，则运行结果如图 2.16 所示。

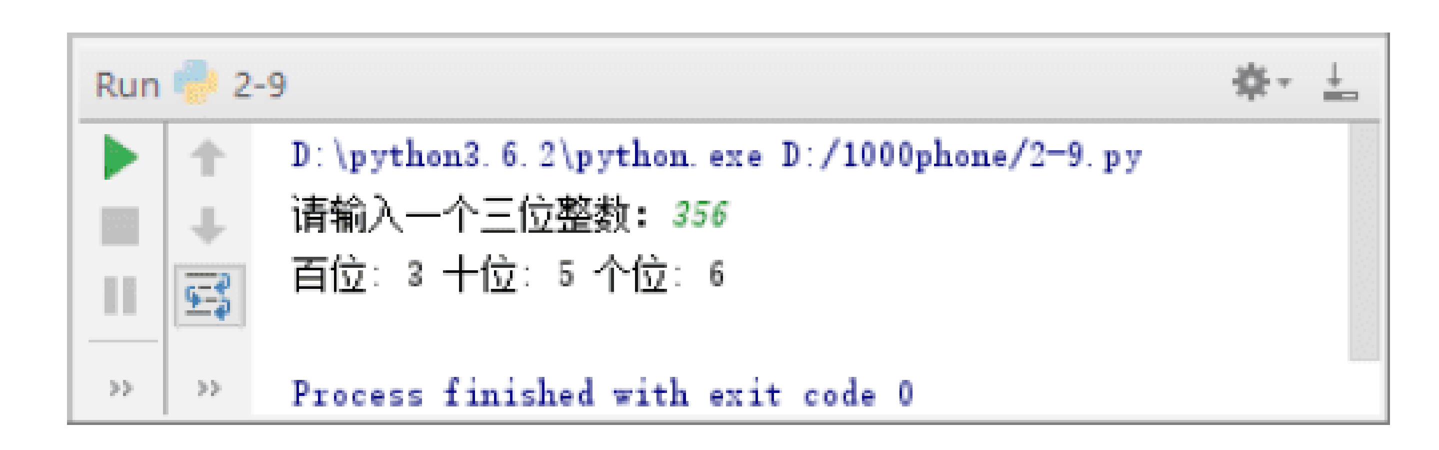


图 2.16 例 2-10 运行结果

暂无须掌握此种用法，等学习完 map()函数后，可再翻阅此处的例题（map()函数将在 8.6 节详细介绍）。

2.5 本章小结

本章主要介绍了 Python 的基本语法，首先讲解标识符与关键字，接着讲解变量与数据类型，最后讲解运算符。本章的知识可能学习起来比较枯燥乏味，却是在将来开发过程中必须掌握的。

2.6 习 题

1．填空题

（1）在 Python 中，单行注释以________开始。

（2）标识符只能以________开头。

（3）在 Python 中，使用________来表示代码块。

（4）若 a = 2，b = 4，则(a or b)的值为________。

（5）布尔型数据只有________和 False 两种值。

2．选择题

（1）下列整型数据用十六进制表示错误的是（ ）。

A．0xac B．0X22 C．0xB D．4fx

（2）下列选项中，不属于数字类型的是（ ）。

A．整型 B．浮点型 C．复数型 D．字符串型

（3）下列选项中，可以用来检测变量数据类型的是（　　）。

A．print()　　B．type()　　C．bin()　　D．int()

（4）若 a＝7，b＝5，下列选项中正确的是（　　）。

A．a//b 的值为 1.4　　B．a/b 的值为 1

C．a**b 的值为 35　　D．a%b 的值为 2

（5）下列表达式中值为 False 的选项是（　　）。

A．0 == False　　B．False == ""

C．0.0 == False　　D．1.0 == True



3．思考题

（1）简述标识符与关键字的区别。

（2）Python 中有哪些数据类型？

4．编程题

编写程序，实现交换两个变量的值。

第 3 章 chapter 3

流程控制语句

本章学习目标

- 掌握 if-else 语句与 if-elif 语句。
- 掌握 while 语句与 for 语句。
- 掌握 break 语句与 continue 语句。

Python 程序设计中流程控制结构包括顺序结构、选择结构和循环结构，它们都是通过控制语句实现的。其中顺序结构不需要特殊的语句，选择结构需要通过条件语句实现，循环结构需要通过循环语句实现。

3.1 条件语句

条件语句可以给定一个判断条件，并在程序执行过程中判断该条件是否成立。程序根据判断结果执行不同的操作，这样就可以改变代码的执行顺序，从而实现更多功能。例如，用户登录某电子邮箱软件，若账号与密码都输入正确，则显示登录成功界面，否则显示登录失败界面，如图 3.1 所示。

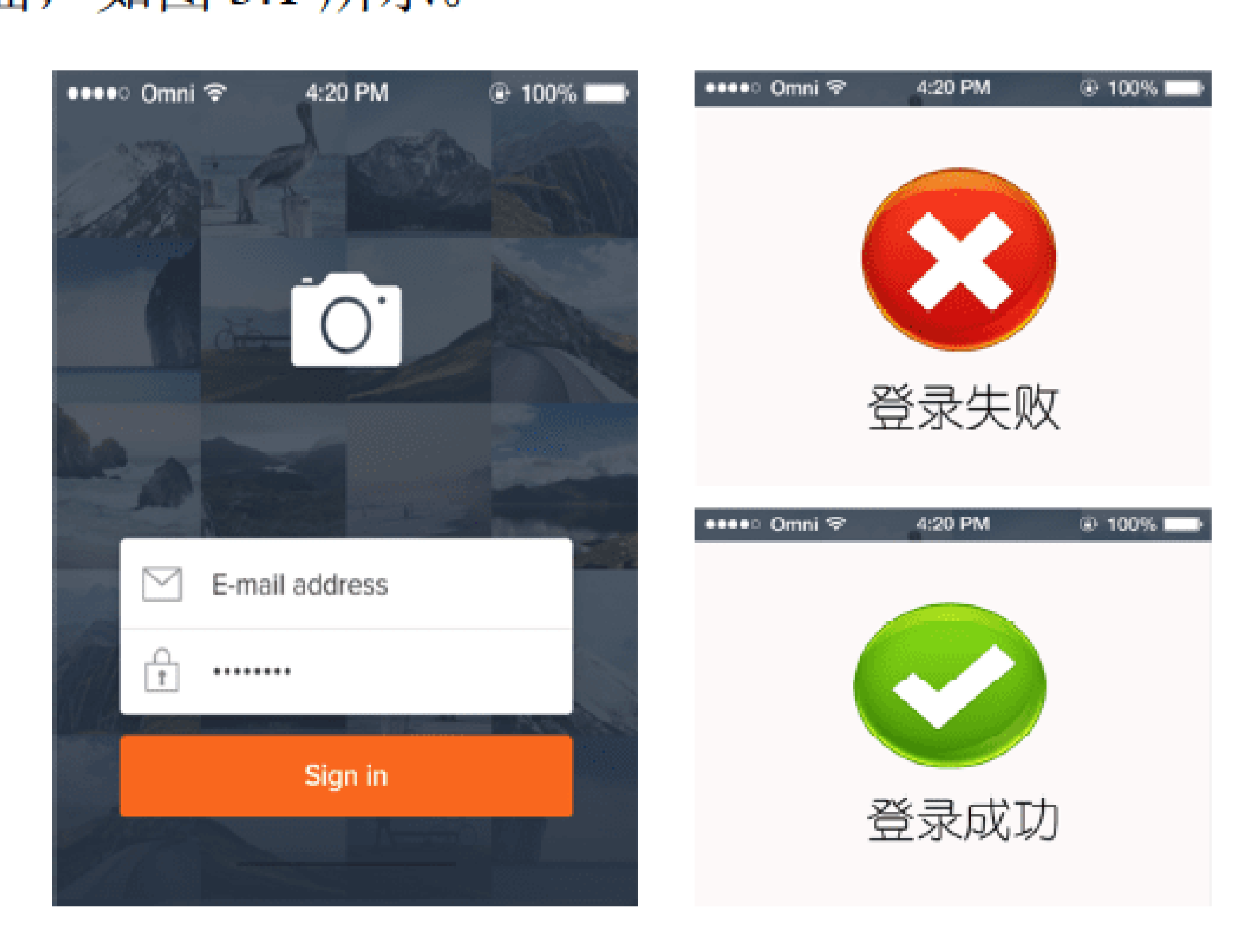


图 3.1 电子邮箱登录界面

Python 中的条件语句有 if 语句、if-else 语句和 if-elif 语句。接下来将针对这些条件语句进行详细讲解。

3.1.1 if 语句

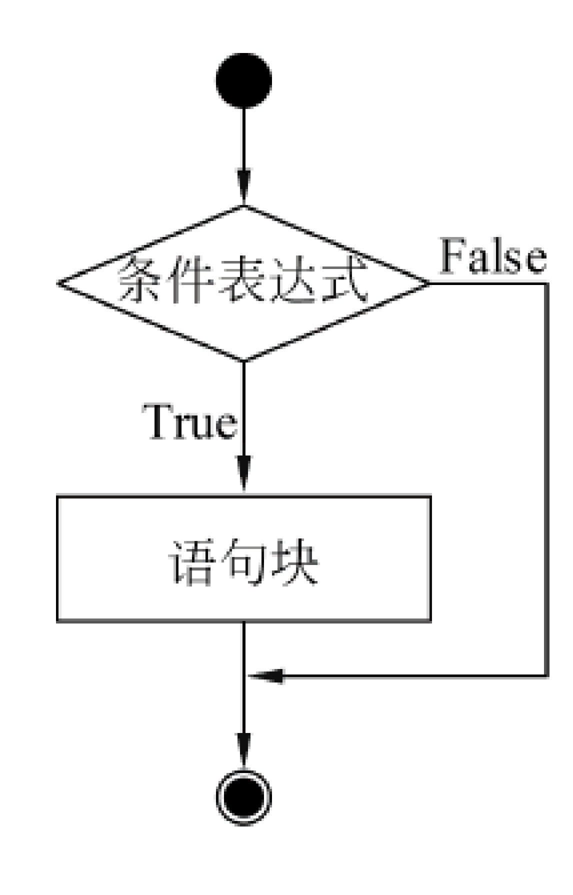


图 3.2 if 语句流程图

if语句用于在程序中有条件地执行某些语句，其语法格式如下：

```
if 条件表达式:
    语句块 # 当条件表达式为 True 时，执行语句块
```

如果条件表达式的值为 True，则执行其后的语句块，否则不执行该语句块。if 语句的执行流程如图 3.2 所示。

接下来演示 if 语句的用法，如例 3-1 所示。

例 3-1 if 语句的用法。

```
1   score = 90
2   if score >= 60:
3       print("真棒！")
4   print("您的分数为%d"%score)
```

运行结果如图 3.3 所示。

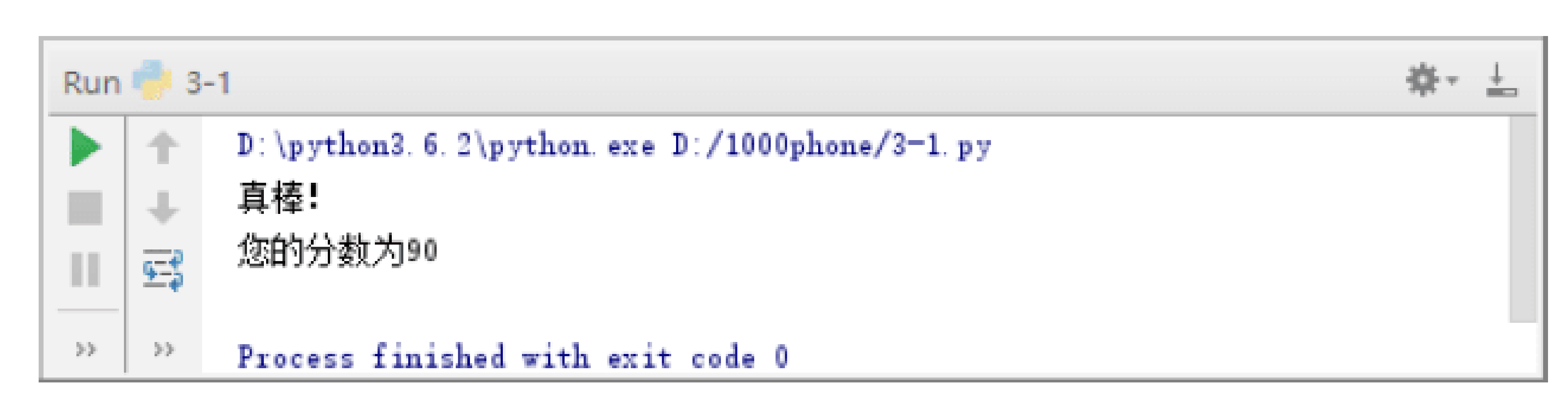


图 3.3 例 3-1 运行结果（一）

如果将变量 score 的值改为 50，则运行结果如图 3.4 所示。

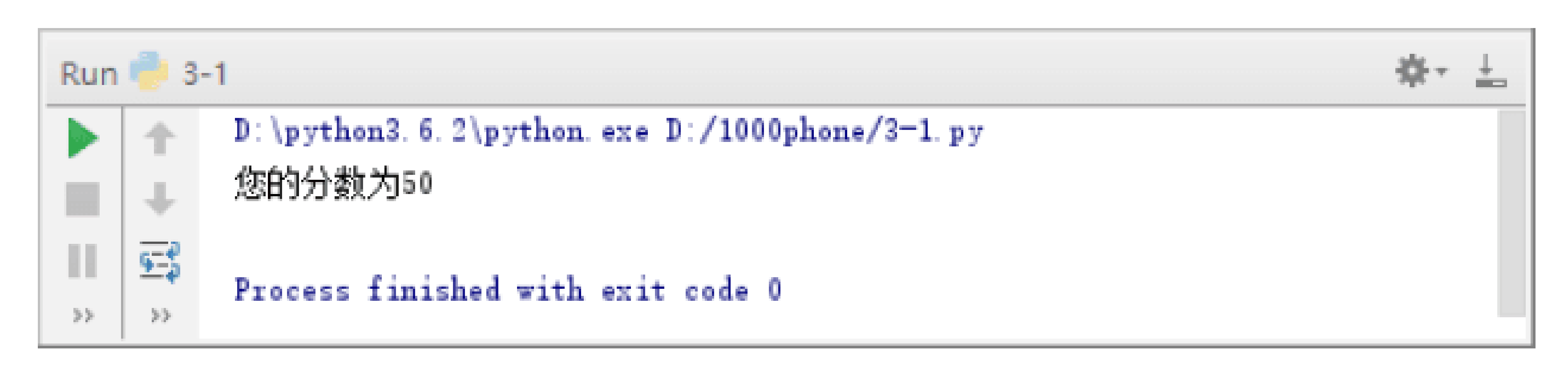


图 3.4 例 3-1 运行结果（二）

在例 3-1 中，第 2 行判断 score 的值是否大于或等于 60。如果 score 的值大于或等于 60，执行第 3 行，否则不执行第 3 行。程序执行完 if 语句后，接着执行第 4 行代码。

3.1.2 if-else 语句

在使用 if 语句时，它只能做到满足条件时执行其后的语句块。如果需要在不满足条件时执行其他语句块，则可以使用 if-else 语句。

if-else 语句用于根据条件表达式的值决定执行哪块代码，其语法格式如下：

```
if 条件表达式:
```

```
    语句块 1  # 当条件表达式为 True 时，执行语句块 1
else:
    语句块 2  # 当条件表达式为 False 时，执行语句块 2
```

如果条件表达式的值为 True，则执行其后的语句块 1，否则执行语句块 2。if-else 语句的执行流程如图 3.5 所示。

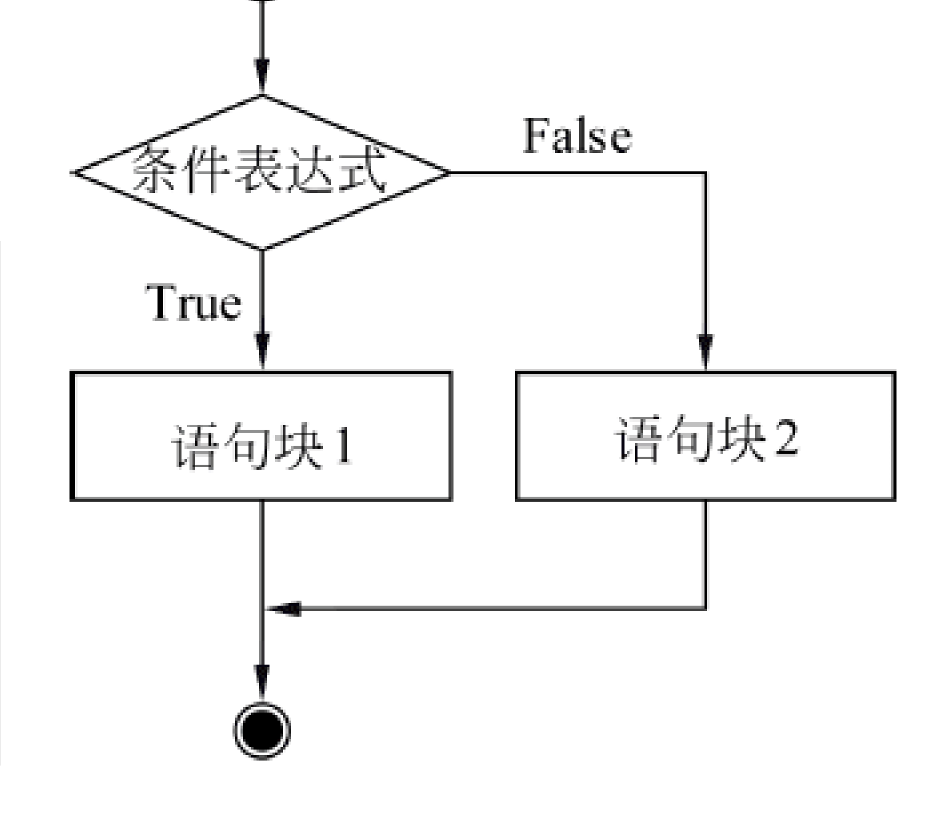


图 3.5　if-else 语句流程图

接下来演示 if-else 语句的用法，如例 3-2 所示。

例 3-2　if-else 语句的用法。

```
1    score = 80
2    if score >= 60:
3        print("真棒！")
4    else:
5        print("加油！")
6    print("您的分数为%d"%score)
```

运行结果如图 3.6 所示。

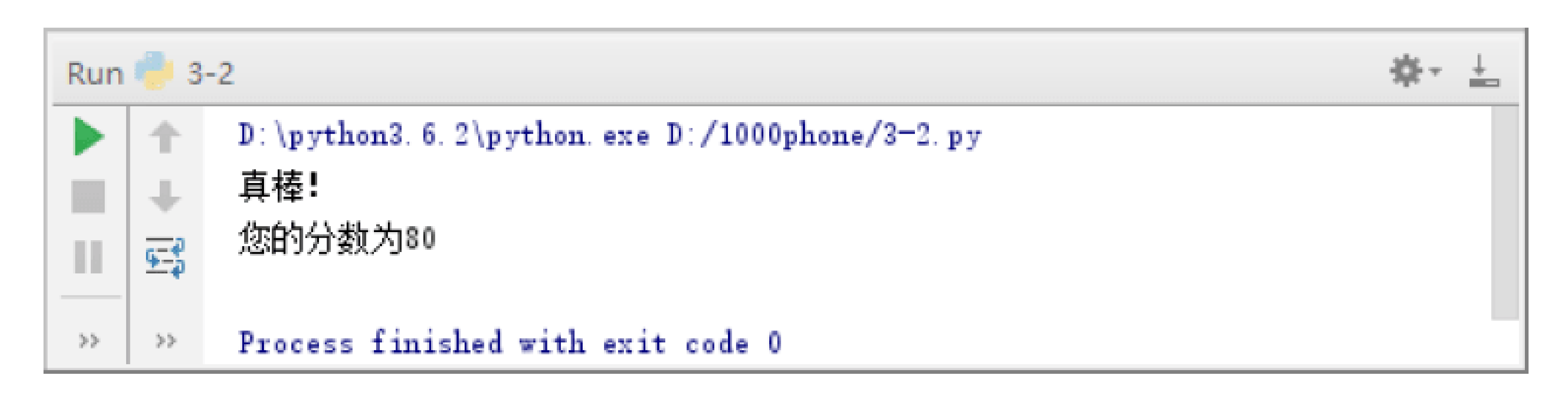


图 3.6　例 3-2 运行结果（一）

如果将变量 score 的值改为 50，则运行结果如图 3.7 所示。

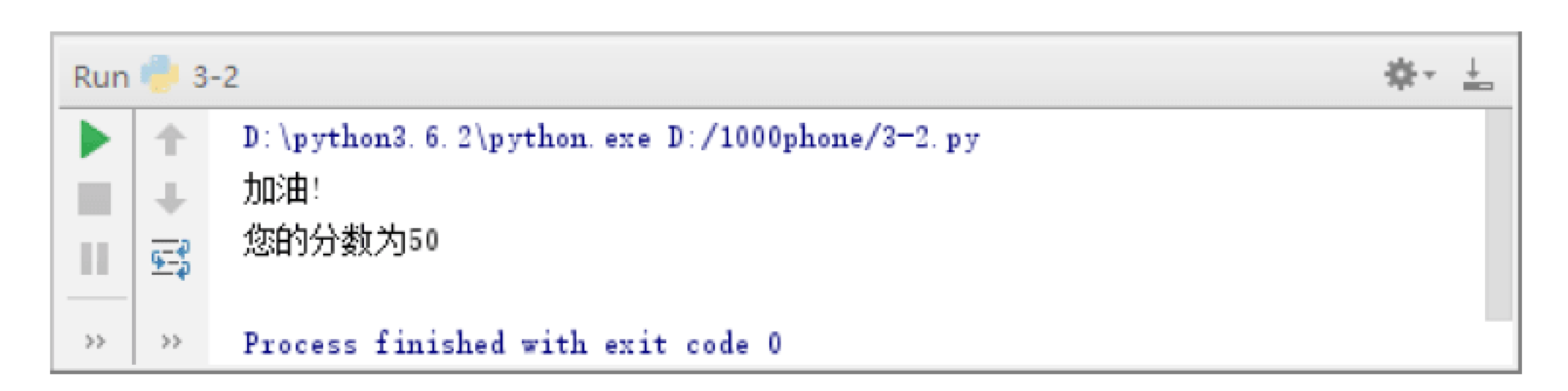


图 3.7　例 3-2 运行结果（二）

在例 3-2 中，第 2 行判断 score 的值是否大于或等于 60，如果 score 的值大于或等于 60，则执行第 3 行，否则执行第 5 行。程序执行完 if-else 语句后，接着执行第 6 行代码。

3.1.3　if-elif 语句

生活中经常需要进行多重判断。例如，考试成绩在 90～100 区间内，称为成绩爆表；在 80～90 区间内，称为成绩优秀；在 60～80 区间内，称为成绩及格；低于 60 的称为成绩堪忧。

在程序中，多重判断可以通过 if-elif 语句实现，其语法格式如下：

```
if 条件表达式 1:
    语句块 1  # 当条件表达式 1 为 True 时,执行语句块 1
elif 条件表达式 2:
    语句块 2  # 当条件表达式 2 为 True 时,执行语句块 2
...
elif 条件表达式 n:
    语句块 n  # 当条件表达式 n 为 True 时,执行语句块 n
```

当执行该语句时，程序依次判断条件表达式的值，当出现某个表达式的值为 True 时，则执行其对应的语句块，然后跳出 if-elif 语句继续执行其后的代码。if-elif 语句的执行流程如图 3.8 所示。

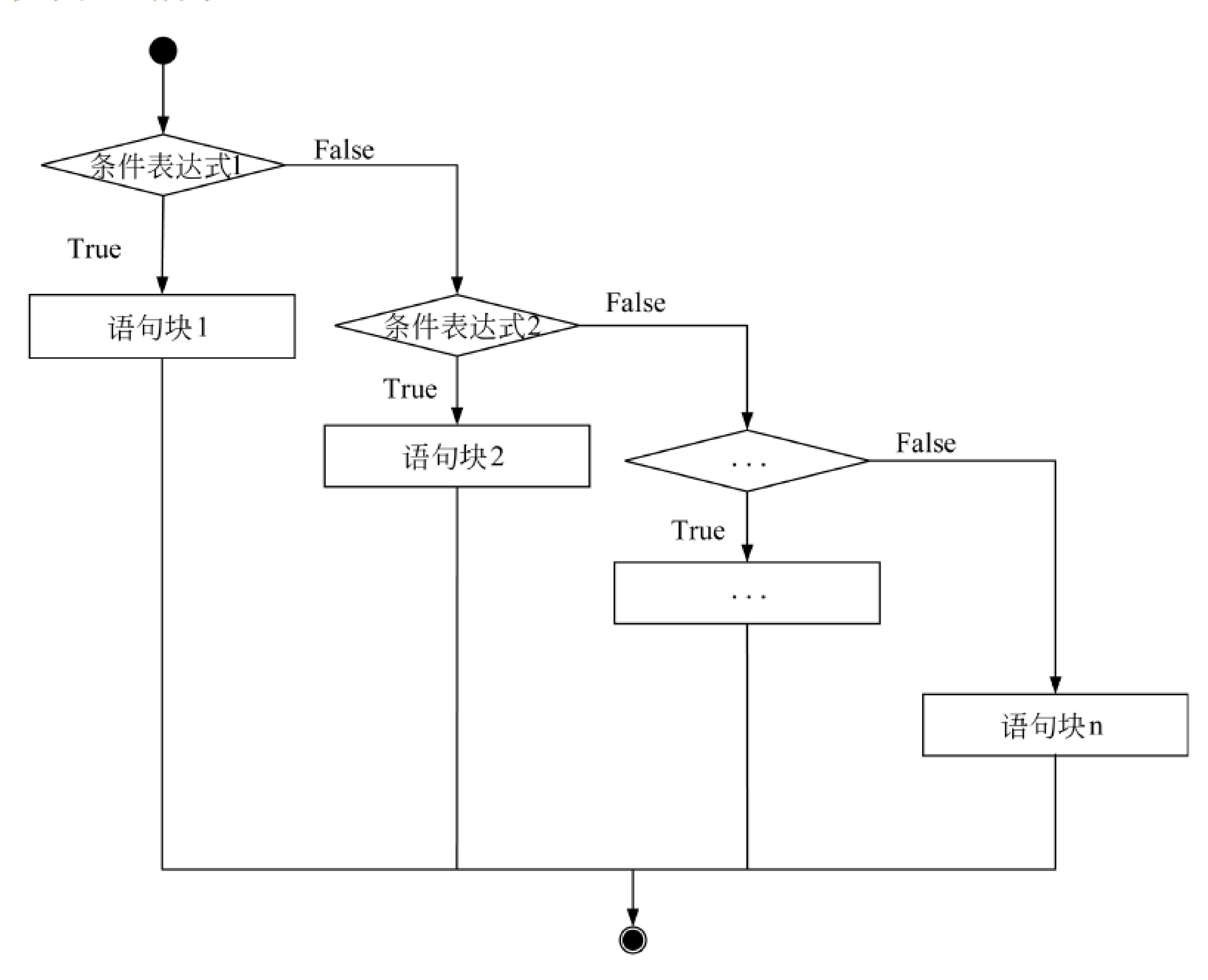


图 3.8　if-elif 语句流程图

接下来演示 if-elif 语句的用法，如例 3-3 所示。

例 3-3　if-elif 语句的用法。

```
score = 80
if 90 <= score <= 100:
    print("成绩爆表！")
elif 80 <= score < 90:
    print("成绩优秀！")
```

```
elif 60 <= score < 80:
    print("成绩及格！")
elif 0 <= score < 60:
    print("成绩堪忧！")
print("您的分数为%d"%score)
```

运行结果如图 3.9 所示。

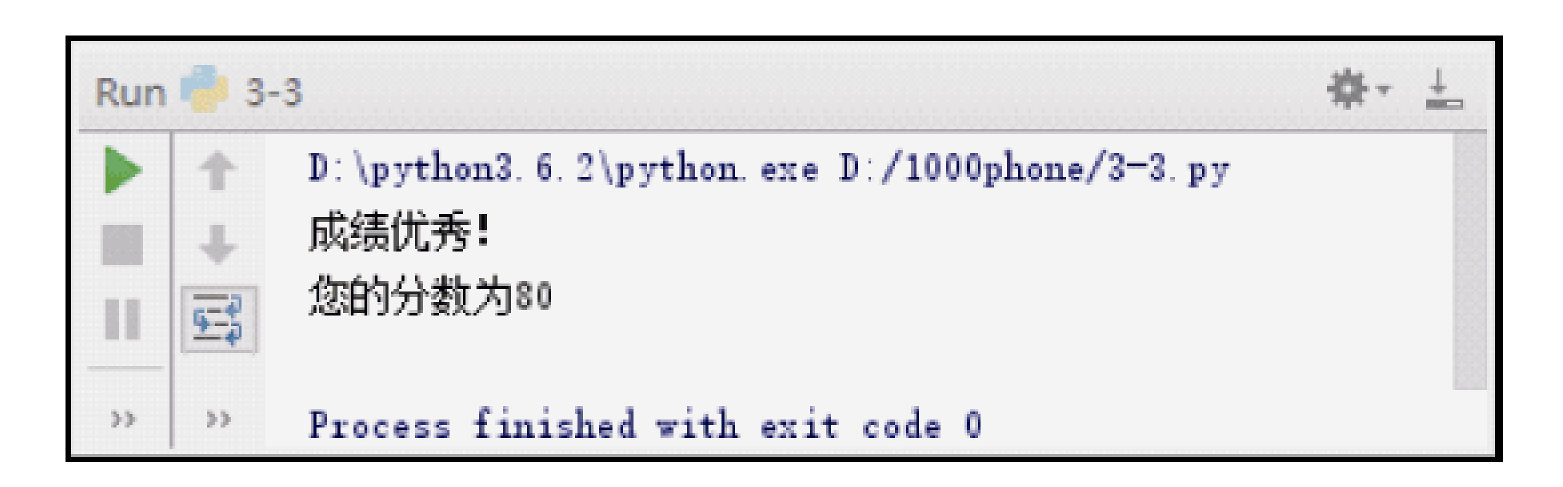


图 3.9　例 3-3 运行结果

在例 3-3 中，首先判断表达式 90 <= score <= 100 的结果为 False，然后接着判断表达式 80 <= score < 90 的结果为 True，则执行其后的语句块。最后，程序跳出 if-elif 语句，执行该语句后面的代码。

此外，if-elif 语句后还可以使用 else 语句，用来表示 if-elif 语句中所有条件不满足时执行的语句块，其语法格式如下：

```
if 条件表达式 1:
    语句块 1 # 当条件表达式 1 为 True 时，执行语句块 1
elif 条件表达式 2:
    语句块 2 # 当条件表达式 2 为 True 时，执行语句块 2
...
else:
    语句块 n # 当以上条件表达式均为 False 时，执行语句块 n
```

接下来演示 if-elif-else 语句的用法，如例 3-4 所示。

例 3-4　if-elif-else 语句的用法。

```
score = 120
if 90 <= score <= 100:
    print("成绩爆表！")
elif 80 <= score < 90:
    print("成绩优秀！")
elif 60 <= score < 80:
    print("成绩及格！")
elif 0 <= score < 60:
    print("成绩堪忧！")
else:
    print("成绩有误！")
print("您的分数为%d"%score)
```

运行结果如图 3.10 所示。

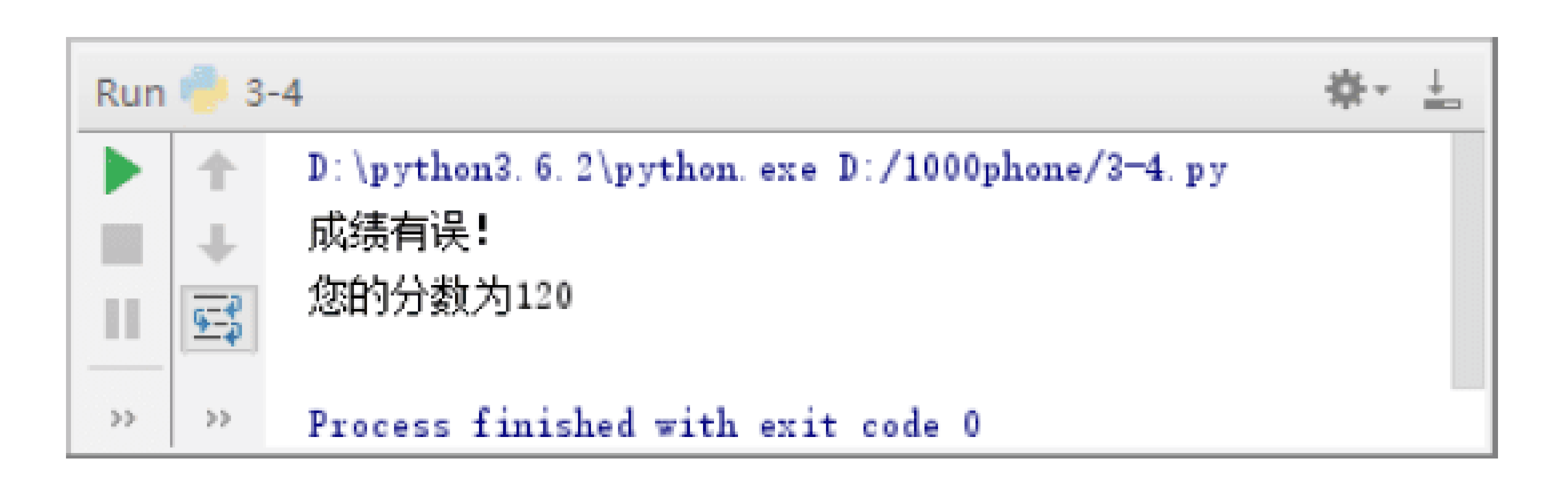


图 3.10　例 3-4 运行结果

在例 3-4 中，if-elif 语句中所有的条件表达式结果都为 False，因此程序将执行 else 语句块。

3.1.4　if 语句嵌套

if 语句嵌套是指 if、if-else 中的语句块可以是 if 或 if-else 语句，其语法格式如下：

```
# if 语句
if 条件表达式 1:
    if 条件表达式 2:    # 嵌套 if 语句
        语句块 2
    if 条件表达式 3:    # 嵌套 if-else 语句
        语句块 3
    else:
        语句块 4
# if-else 语句
if 条件表达式 1:
    if 条件表达式 2:    # 嵌套 if 语句
        语句块 2
else:
    if 条件表达式 3:    # 嵌套 if-else 语句
        语句块 3
    else:
        语句块 4
```

注意 if 语句嵌套有多种形式，在实际编程时需灵活使用。接下来演示 if 嵌套语句的使用，如例 3-5 所示。

例 3-5　if 嵌套语句的用法。

```
a, b, c = 5, 8, 3
if a >= b:
    if a >= c:
        print("a、b、c 中最大的值为%d"%a)
    else:
        print("a、b、c 中最大的值为%d"%c)
```

```
7    else:
8        if b >= c:
9            print("a、b、c 中最大的值为%d"%b)
10       else:
11           print("a、b、c 中最大的值为%d"%c)
```

运行结果如图 3.11 所示。

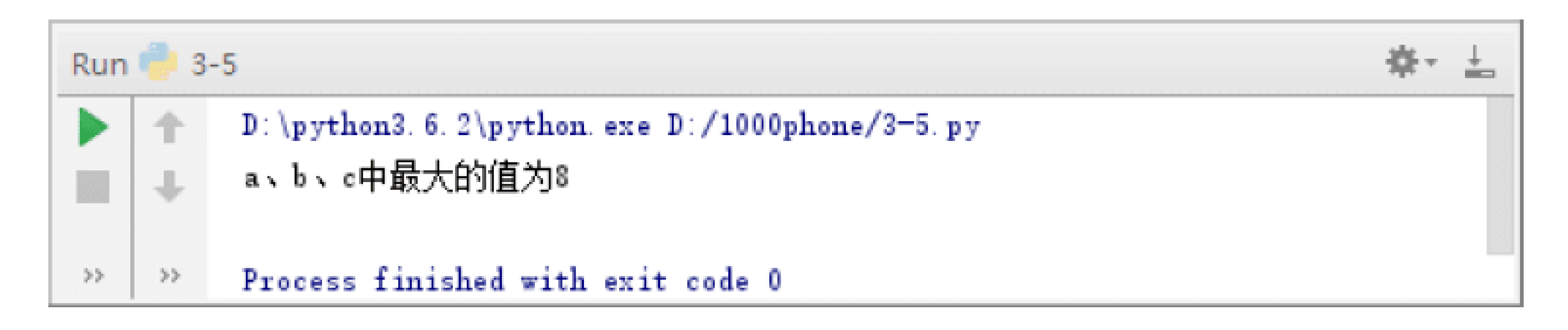


图 3.11　例 3-5 运行结果

在例 3-5 中，程序的功能是输出 a、b、c 中最大的值。第 2～6 行为 a 大于或等于 b 的情形，第 7～11 行为 a 小于 b 的情形。

3.2　循环语句

循环的意思就是让程序重复地执行某些语句。在实际应用中，当碰到需要多次重复地执行一个或多个任务时，可考虑使用循环语句来解决。循环语句的特点是在给定条件成立时，重复执行某个程序段。通常称给定条件为循环条件，称反复执行的程序段为循环体。

3.2.1　while 语句

在 while 语句中，当条件表达式为 True 时，就重复执行语句块；当条件表达式为 False 时，就结束执行语句块。while 语句的语法格式如下：

```
while 条件表达式:
    语句块 # 此处语句块也称循环体
```

while 语句中循环体是否执行，取决于条件表达式是否为 True。当条件表达式为 True 时，循环体就会被执行，循环体执行完毕后继续判断条件表达式，如果条件表达式为 True，则会继续执行，直到条件表达式为 False 时，整个循环过程才会执行结束。while 语句的执行流程如图 3.12 所示。

图 3.12　while 循环语句流程图

接下来演示 while 语句的用法，如例 3-6 所示。

例 3-6　while 语句的用法。

```
i, sum = 1, 0
while i < 101:
```

```
3       sum += i
4       i += 1
5   print("1 + 2 + … + 100 = %d"%sum)
```

运行结果如图 3.13 所示。

```
Run  3-6
D:\python3.6.2\python.exe D:/1000phone/3-6.py
1 + 2 + · · · + 100 = 5050

Process finished with exit code 0
```

图 3.13　例 3-6 运行结果

在例 3-6 中，程序功能是实现 1～100 的累加和。当 i=1 时，i<101，此时执行循环体语句块，sum 为 1，i 为 2。当 i=2 时，i<101，此时执行循环体语句块，sum 为 3，i 为 3。以此类推，直到 i=101，不满足循环条件，此时程序执行第 5 行代码。

3.2.2　for 语句

for 语句可以循环遍历任何序列中的元素，如列表、元组、字符串等，其语法格式如下：

```
for 元素 in 序列:
    语句块
```

其中，for、in 为关键字，for 后面是每次从序列中取出的一个元素。接下来演示 for 语句的用法，如例 3-7 所示。

例 3-7　for 语句的用法。

```
for word in "Python":
    print(word)
```

运行结果如图 3.14 所示。

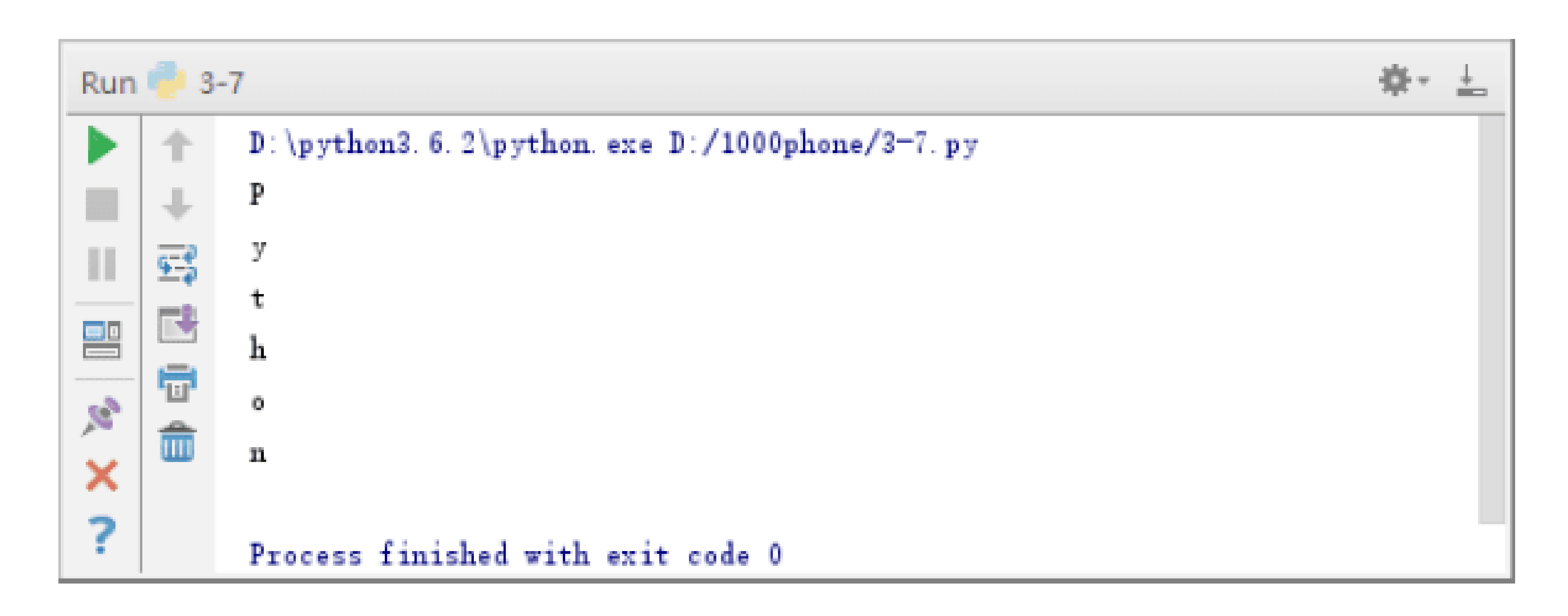


图 3.14　例 3-7 运行结果

在例 3-7 中，for 语句将字符串中的每个字符逐个赋值给 word，然后通过 print()函数输出。

当需要遍历数字序列时，可以使用 range()函数，它会生成一个数列，接下来演示其用法，如例 3-8 所示。

例 3-8 range()函数的用法。

```
sum = 0
for i in range(1, 101):
    sum += i
print("1 + 2 + ••• + 100 = %d"%sum)
```

运行结果如图 3.15 所示。

```
Run 3-8
D:\python3.6.2\python.exe D:/1000phone/3-8.py
1 + 2 + ・・・ + 100 = 5050

Process finished with exit code 0
```

图 3.15 例 3-8 运行结果

在例 3-8 中，通过 range()函数可以生成一个 1～100 组成的数字序列，当使用 for 遍历时，依次从这个数字序列中取值。

3.2.3 while 与 for 嵌套

while 语句中可以嵌套 while 语句或 for 语句。接下来演示 while 语句中嵌套 while 语句，如例 3-9 所示。

例 3-9 while 语句中嵌套 while 语句。

```
i = 1
while i < 10:
    j = 1
    while j <= i:
        print("%d×%d = %-3d"%(i, j, i*j), end = ' ')
        j += 1
    i += 1
    print(end = '\n')
```

运行结果如图 3.16 所示。

```
Run 3-9
D:\python3.6.2\python.exe D:/1000phone/3-9.py
1×1 = 1
2×1 = 2   2×2 = 4
3×1 = 3   3×2 = 6   3×3 = 9
4×1 = 4   4×2 = 8   4×3 = 12  4×4 = 16
5×1 = 5   5×2 = 10  5×3 = 15  5×4 = 20  5×5 = 25
6×1 = 6   6×2 = 12  6×3 = 18  6×4 = 24  6×5 = 30  6×6 = 36
7×1 = 7   7×2 = 14  7×3 = 21  7×4 = 28  7×5 = 35  7×6 = 42  7×7 = 49
8×1 = 8   8×2 = 16  8×3 = 24  8×4 = 32  8×5 = 40  8×6 = 48  8×7 = 56  8×8 = 64
9×1 = 9   9×2 = 18  9×3 = 27  9×4 = 36  9×5 = 45  9×6 = 54  9×7 = 63  9×8 = 72  9×9 = 81

Process finished with exit code 0
```

图 3.16 例 3-9 运行结果

在例 3-9 中，第 2～8 行为外层 while 循环，第 4～6 行为内层 while 循环，其中变量 i 控制行，变量 j 控制列，乘法表中的每一项可以表示为 i×j = i*j。

接下来演示 while 语句中嵌套 for 语句，如例 3-10 所示。

例 3-10 while 语句中嵌套 for 语句。

```
1  i = 1
2  while i < 10:
3      for j in range(1, i + 1):
4          print("%d×%d = %-3d"%(i, j, i*j), end = ' ')
5      i+= 1
6      print(end = '\n')
```

运行结果如图 3.17 所示。

```
Run 3-10
D:\python3.6.2\python.exe D:/1000phone/3-10.py
1×1 = 1
2×1 = 2   2×2 = 4
3×1 = 3   3×2 = 6   3×3 = 9
4×1 = 4   4×2 = 8   4×3 = 12  4×4 = 16
5×1 = 5   5×2 = 10  5×3 = 15  5×4 = 20  5×5 = 25
6×1 = 6   6×2 = 12  6×3 = 18  6×4 = 24  6×5 = 30  6×6 = 36
7×1 = 7   7×2 = 14  7×3 = 21  7×4 = 28  7×5 = 35  7×6 = 42  7×7 = 49
8×1 = 8   8×2 = 16  8×3 = 24  8×4 = 32  8×5 = 40  8×6 = 48  8×7 = 56  8×8 = 64
9×1 = 9   9×2 = 18  9×3 = 27  9×4 = 36  9×5 = 45  9×6 = 54  9×7 = 63  9×8 = 72  9×9 = 81

Process finished with exit code 0
```

图 3.17 例 3-10 运行结果

在例 3-10 中，第 2～6 行为外层 while 循环，第 3～4 行为内层 for 循环，其中变量 i 控制行，变量 j 控制列，乘法表中的每一项可以表示为 i×j = i*j。

此外，for 语句中可以嵌套 while 语句或 for 语句。接下来演示 for 语句中嵌套 while 语句，如例 3-11 所示。

例 3-11 for 语句中嵌套 while 语句。

```
1  for i in range(1, 10):
2      j = 1
3      while j <= i:
4          print("%d×%d = %-3d"%(i, j, i*j), end = ' ')
5          j += 1
6      print(end = '\n')
```

运行结果如图 3.18 所示。

在例 3-11 中，第 1～6 行为外层 for 循环，第 3～5 行为内层 while 循环，其中变量 i 控制行，变量 j 控制列，乘法表中的每一项可以表示为 i×j = i*j。

接下来演示 for 语句中嵌套 for 语句，如例 3-12 所示。

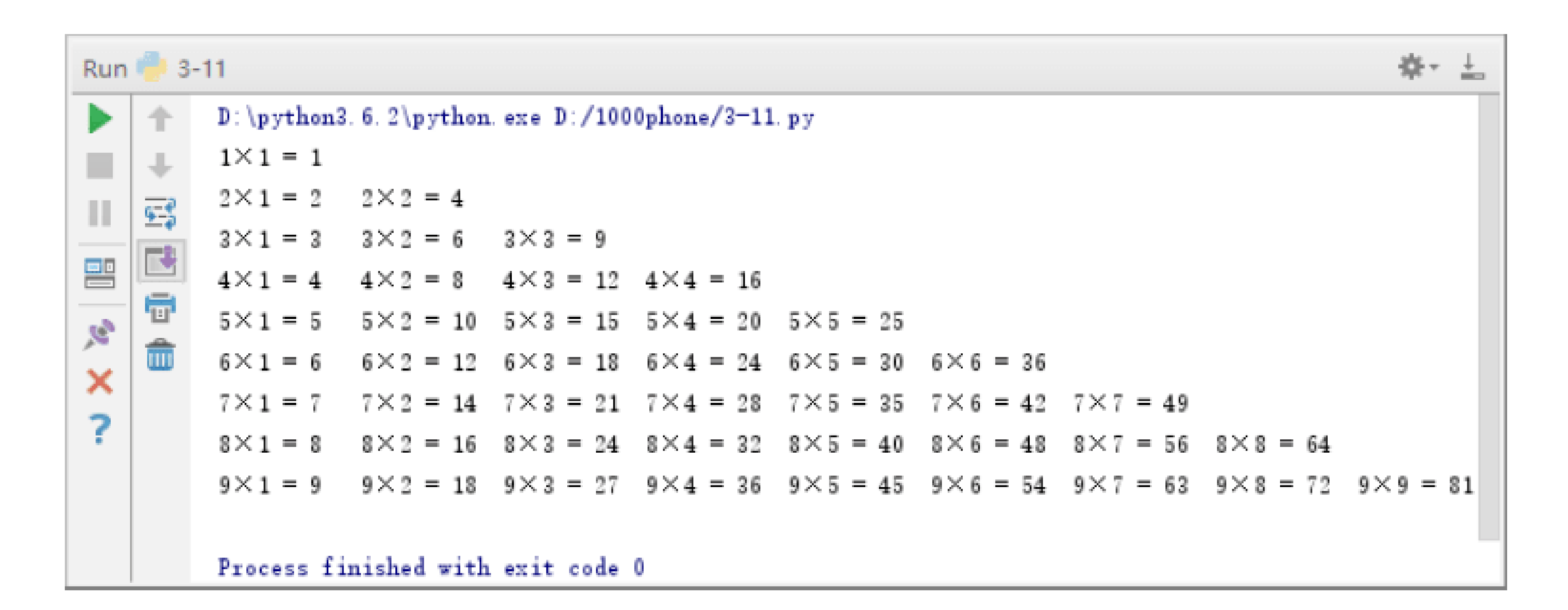


图 3.18　例 3-11 运行结果

例 3-12　for 语句中嵌套 for 语句。

```
1    for i in range(1, 10):
2        for j in range(1, i + 1):
3            print("%d×%d = %-3d"%(i, j, i*j), end = ' ')
4        print(end = '\n')
```

运行结果如图 3.19 所示。

```
Run 3-12
D:\python3.6.2\python.exe D:/1000phone/3-12.py
1×1 = 1
2×1 = 2   2×2 = 4
3×1 = 3   3×2 = 6   3×3 = 9
4×1 = 4   4×2 = 8   4×3 = 12  4×4 = 16
5×1 = 5   5×2 = 10  5×3 = 15  5×4 = 20  5×5 = 25
6×1 = 6   6×2 = 12  6×3 = 18  6×4 = 24  6×5 = 30  6×6 = 36
7×1 = 7   7×2 = 14  7×3 = 21  7×4 = 28  7×5 = 35  7×6 = 42  7×7 = 49
8×1 = 8   8×2 = 16  8×3 = 24  8×4 = 32  8×5 = 40  8×6 = 48  8×7 = 56  8×8 = 64
9×1 = 9   9×2 = 18  9×3 = 27  9×4 = 36  9×5 = 45  9×6 = 54  9×7 = 63  9×8 = 72  9×9 = 81

Process finished with exit code 0
```

图 3.19　例 3-12 运行结果

在例 3-12 中，第 1～4 行为外层 for 循环，第 2～3 行为内层 for 循环，其中变量 i 控制行，变量 j 控制列，乘法表中的每一项可以表示为 i×j = i*j。

3.2.4　break 语句

break 语句可以使程序立即退出循环，转而执行该循环外的下一条语句。如果 break 语句出现在嵌套循环的内层循环中，则 break 语句只会跳出当前层的循环。

接下来演示 break 语句的用法，如例 3-13 所示。

例 3-13　break 语句的用法。

```
i = 0
while True:
    i += 1
    print("第%d 次循环开始"%i)
```

```
    if i == 3:
        break
    print("第%d 次循环结束"%i)
print("整个循环结束")
```

运行结果如图 3.20 所示。

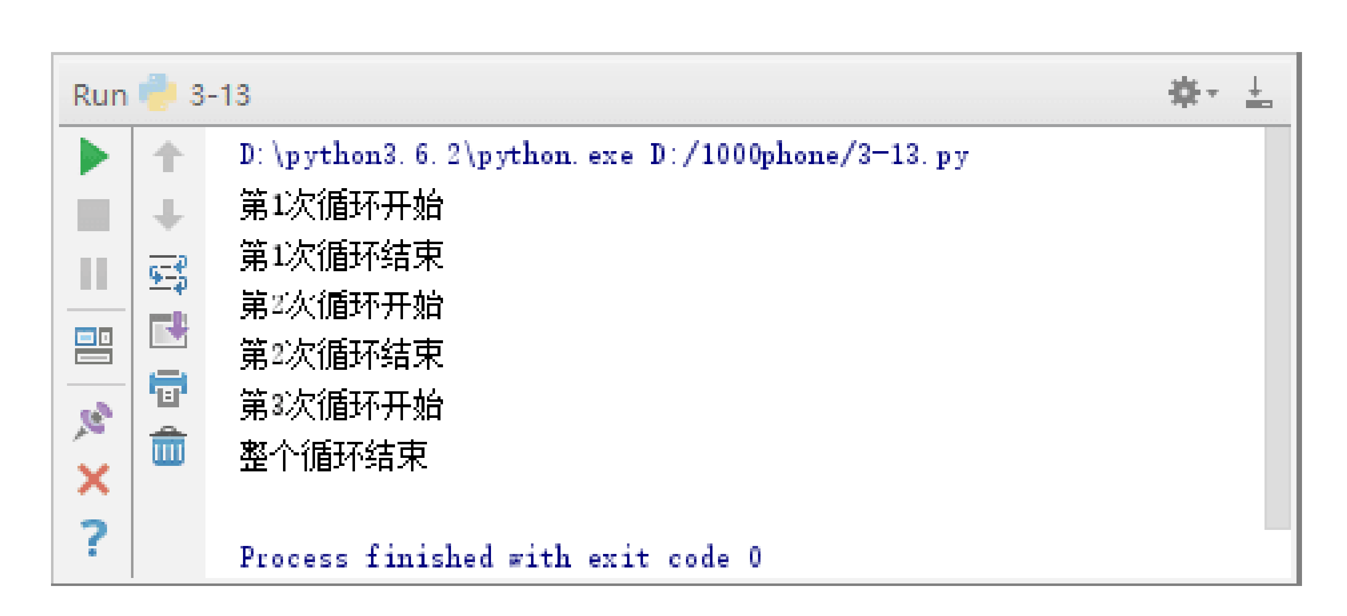


图 3.20　例 3-13 运行结果

在例 3-13 中，while 语句中增加 if 条件语句。当 i 为 3 时，程序跳出循环。如果没有此 if 语句，程序会一直执行循环，直到计算机崩溃，这种循环称为无限循环。

3.2.5　continue 语句

continue 语句用于跳过当次循环体中剩余的语句，然后进行下一次循环。接下来演示其用法，如例 3-14 所示。

例 3-14　continue 语句的用法。

```
i = 0
while i < 3:
    i += 1
    print("第%d 次循环开始"%i)
    if i == 2:
        continue
    print("第%d 次循环结束"%i)
print("整个循环结束")
```

运行结果如图 3.21 所示。

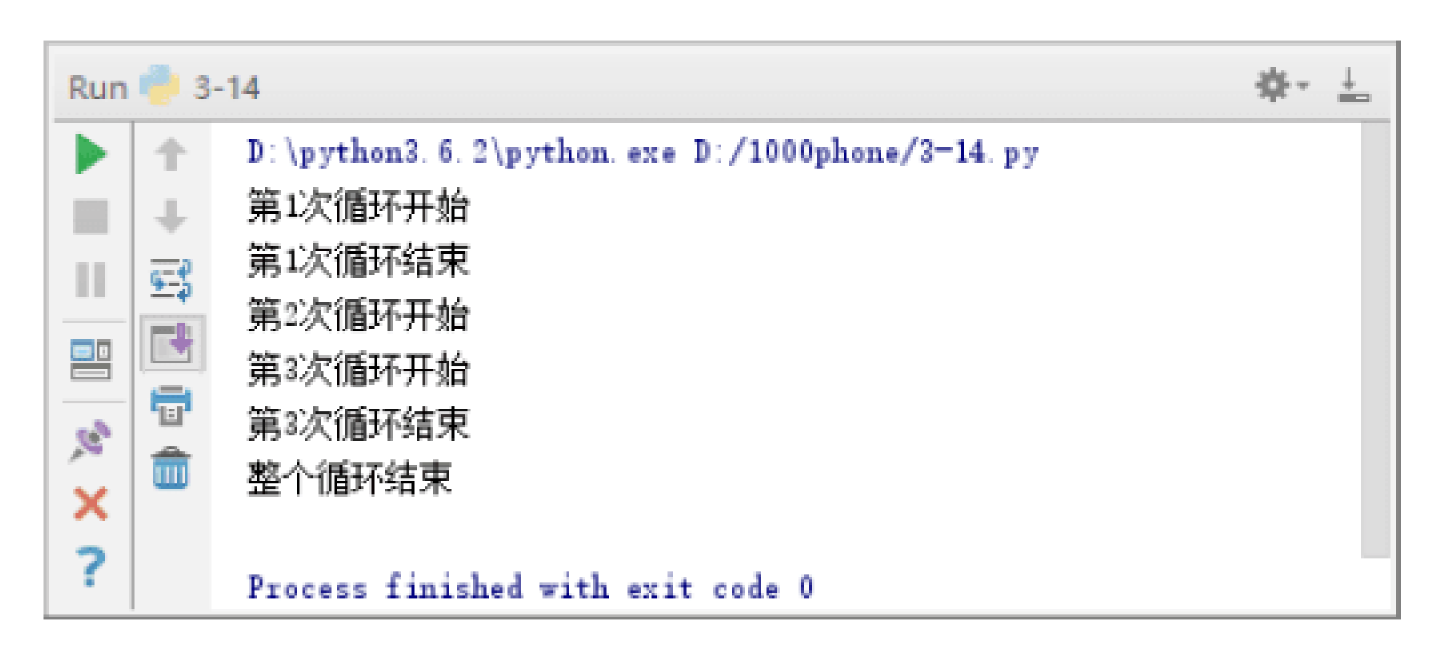


图 3.21　例 3-14 运行结果

在例 3-14 中，while 语句中增加了 if 条件语句。当 i 为 2 时，程序跳出第 2 次循环，接着开始执行第 3 次循环。

3.2.6　else 语句

else 语句除了可以与 if 语句搭配使用外，还可以与 while 语句、for 语句搭配使用，当条件不满足时执行 else 语句块，它只在循环结束后执行。接下来演示 for 语句搭配 else 语句用法，如例 3-15 所示。

例 3-15　for 语句搭配 else 语句的用法。

```
for n in range(1, 3):
    print("第%d 次循环"%n)
else:
    print("循环结束")
```

运行结果如图 3.22 所示。

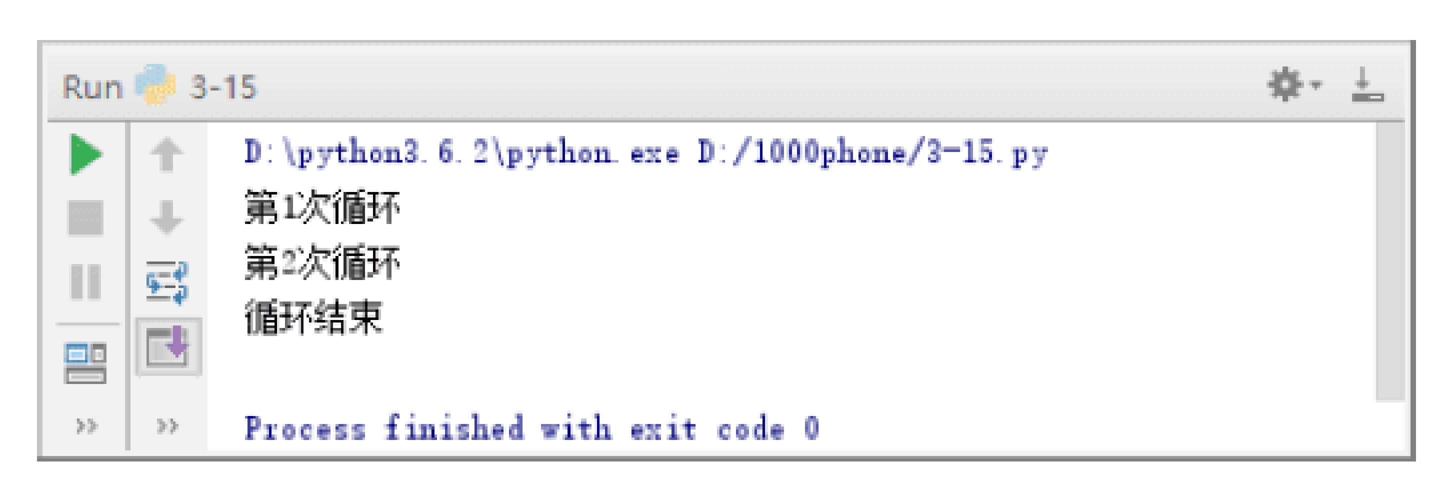


图 3.22　例 3-15 运行结果

在例 3-15 中，for 语句后添加 else 语句，从程序运行结果可看出，程序执行完 for 语句后，接着执行 else 语句。

此处需注意，while 语句或 for 语句中有 break 语句时，程序将会跳过 while 语句或 for 语句后的 else 语句，接下来演示这种情形，如例 3-16 所示。

例 3-16　for 语句中存在 break 语句。

```
for n in range(1, 4):
    print("第%d 次循环"%n)
    if n == 2:
        break
else:
    print("循环结束")
print("程序结束")
```

运行结果如图 3.23 所示。

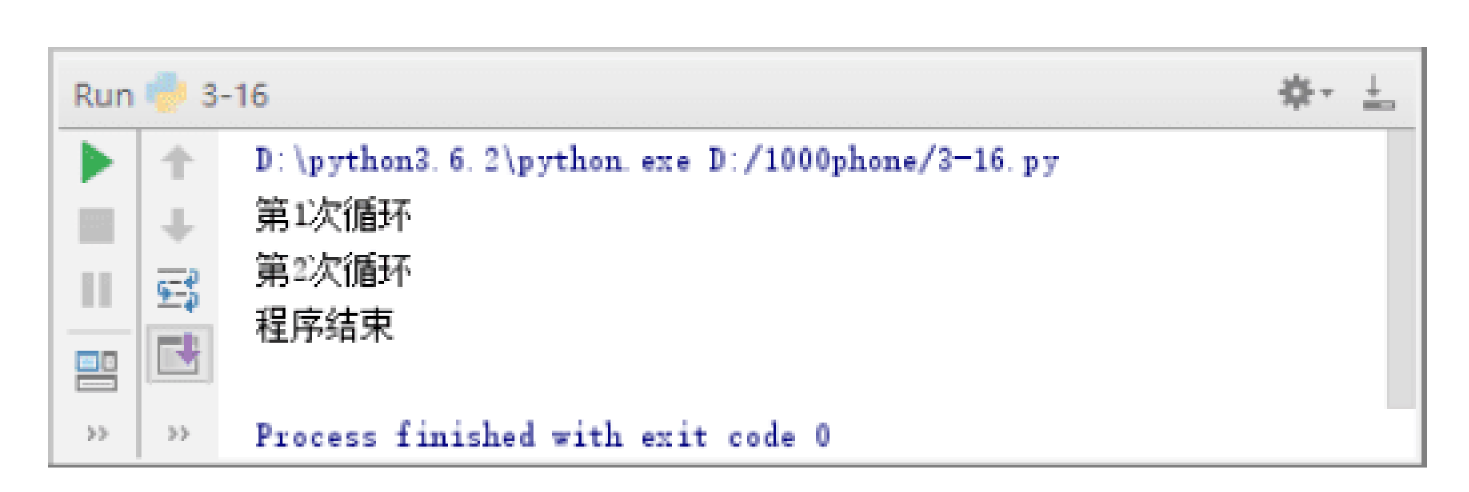


图 3.23　例 3-16 运行结果

在例 3-16 中，for 语句中出现 break 语句。当 n 为 2 时，程序跳出 for 循环，并且没有执行 else 语句。

3.2.7 pass 语句

在编写一个程序时，如果对部分语句块还没有编写思路，这时可以用 pass 语句来占位。它可以当作一个标记，表示未完成的代码块。

接下来演示 pass 语句的用法，如例 3-17 所示。

例 3-17 pass 语句的用法。

```
for n in range(1, 3):
    pass
    print("暂时没思路")
print("程序结束")
```

运行结果如图 3.24 所示。

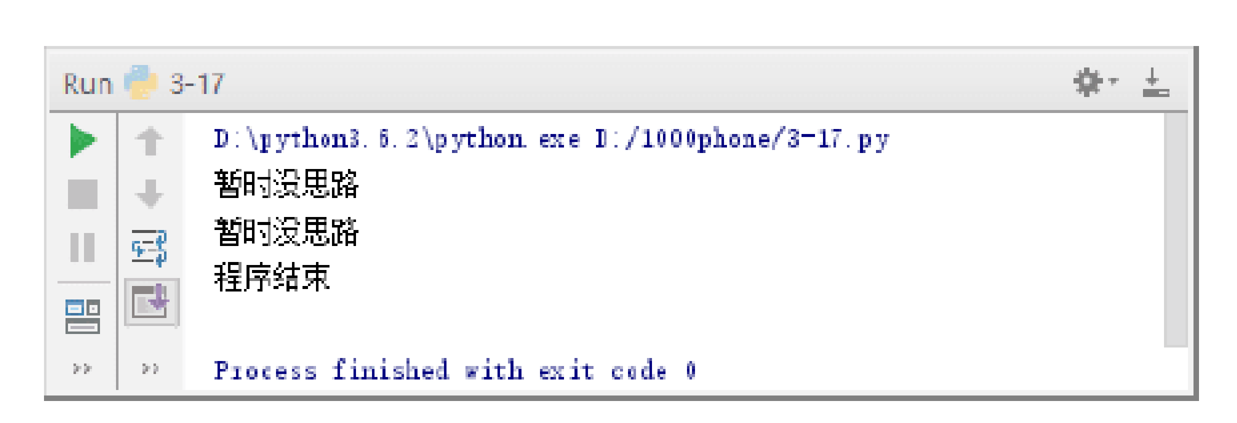


图 3.24 例 3-17 运行结果

在例 3-17 中，当执行 pass 语句时，程序会忽略该语句，按顺序执行其他语句。

3.3 小 案 例

3.3.1 案例一

“鸡兔同笼问题”是我国古算书《孙子算经》中著名的数学问题，其内容是：“今有雉（鸡）兔同笼，上有三十五头，下有九十四足，问雉兔各几何？”具体实现如例 3-18 所示。

例 3-18 鸡兔同笼问题。

```
1   for chicken in range(0, 36):
2       if 2 * chicken + (35 - chicken) * 4 == 94:
3           print('小鸡:', chicken, ' 小兔:', 35 - chicken)
```

运行结果如图 3.25 所示。

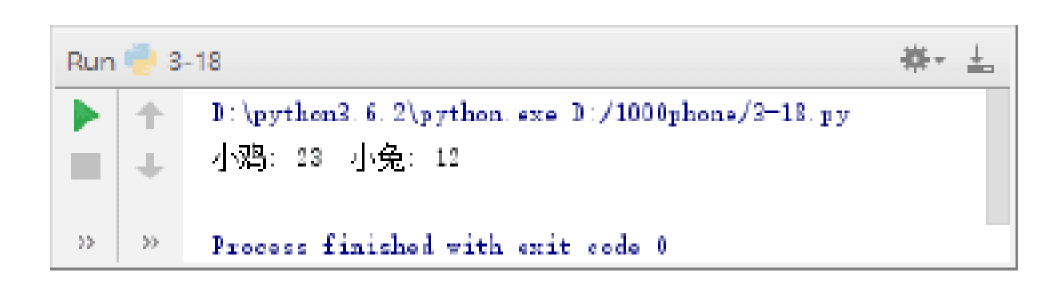


图 3.25 例 3-18 运行结果

在例 3-18 中，程序通过 for 循环依次判断 0～35 的整数是否满足第 2 行 if 语句的条

件。如果满足该条件，程序就计算出鸡兔同笼的答案。

3.3.2 案例二

程序输入若干个学生某门课程成绩，求出这些学生成绩的平均值、最大值和最小值，具体实现如例 3-19 所示。

例 3-19 求学生成绩的平均值、最大值和最小值。

```
num, sum, max, min = 0, 0, 0, 100
while True:
    str = input('请输入第%d 位学生成绩:'%(num + 1))
    if str == 'Q':
        print('输入结束! ')
        break
    score = int(str)
    if score < 0 or score > 100:
        print('输入有误,重新输入!')
        continue
    sum += score
    num += 1
    if score > max:
        max = score
    if score < min:
        min = score
print('平均成绩:', sum * 1.0 / num)
print('最大值:', max, '最小值:', min)
```

运行结果如图 3.26 所示。

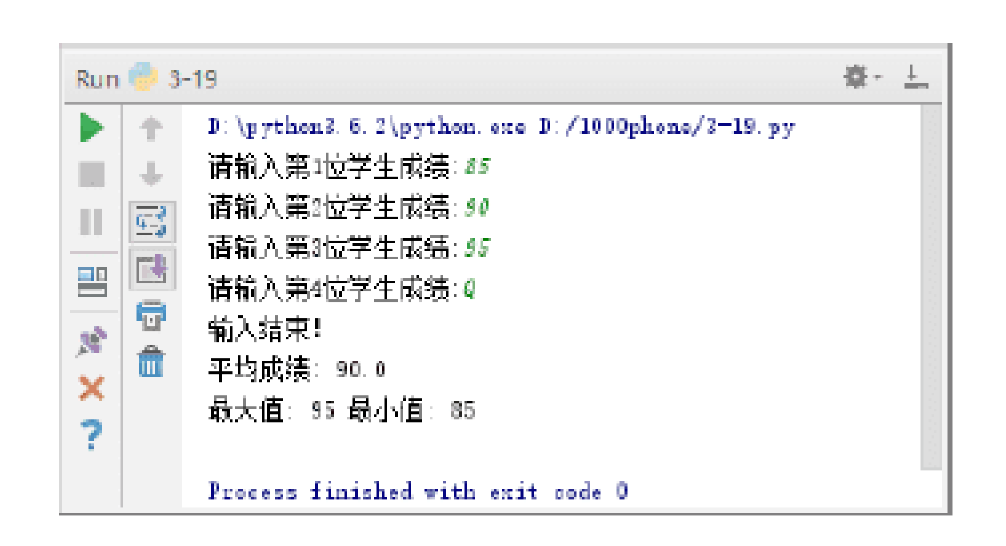


图 3.26 例 3-19 运行结果

在例 3-19 中，当输入 Q 时，程序结束循环。当输入的数字小于 0 或大于 100 时，程序结束本次循环。当输入的分数比最大值大时，程序将输入的分数赋值给最大值。当输入的分数比最小值小时，程序将输入的分数赋值给最小值。

3.4 本 章 小 结

通过本章的学习，大家需熟练掌握条件语句与循环语句的使用。当需要对某种条件进

行判断，并且为真或为假时分别执行不同的语句时，可以使用 if-else 语句。当需要检测的条件很多，可以使用 if-elif 语句。当需重复执行某些语句，并且能够确定执行的次数时，可以使用 for 语句；假如不能确定执行的次数，可以使用 while 语句。另外，continue 语句可以使当次循环结束，并从循环的开始处继续执行下次循环，break 语句会使循环直接结束。

3.5 习　　题

1．填空题

（1）________语句表示占位。
（2）________语句是 if 语句与 else 语句的组合。
（3）________语句可以使程序立即退出循环，转而执行该循环外的下一条语句。
（4）________语句用于跳过当前循环体中剩余语句，然后继续进行下一次循环。
（5）________语句用于进行多重判断。

2．选择题

（1）每个 if 条件后需要使用（　　）。
A．冒号　　B．分号　　C．中括号　　D．大括号
（2）下列语句不能嵌套自身的是（　　）。
A．if 语句　　B．else 语句　　C．while 语句　　D．for 语句
（3）下列选项中，可以生成 1～5 的数字序列的是（　　）。
A．range(0,5)　　B．range(1,5)　　C．range(1,6)　　D．range(0,6)
（4）下列语句不能单独使用的是（　　）。
A．if 语句　　B．elif 语句　　C．if-else 语句　　D．for 语句
（5）对于 for n in range(0,3):print(n)，共循环（　　）次。
A．4　　B．2　　C．0　　D．3



3．思考题

（1）简述 else 语句可以与哪些语句配合使用。
（2）简述 break 语句与 continue 语句的区别。

4．编程题

编写程序输出 1～100 的质数。

第 4 章 chapter 4

字 符 串

本章学习目标

- 掌握字符串的 3 种表现形式。
- 掌握字符串的输入与输出。
- 掌握字符串的索引与切片。
- 了解字符串的运算。
- 熟悉字符串常用函数。

字符串是由若干子串组成的序列，其主要用来表示文本，例如登录网站时输入的用户名与密码等。灵活地使用与处理字符串，对于 Python 程序员来说是非常重要的。

4.1 字符串简介

在汉语中，将若干个字连起来就是一个字符串，例如“千锋教育”就是一个由 4 个汉字组成的字符串。在程序中，字符串是由若干字符组成的序列。

4.1.1 字符串的概念

在前面的章节中，大家已接触过简单字符串，Python 中的字符串以引号包含为标识，具体有 3 种表现形式。

1. 使用单引号标识字符串

使用单引号标识的字符串中不能包含单引号，具体如下所示：

```
'xiaoqian'
'666'
'小千说:"坚持到感动自己，拼搏到无能为力"。'
```

2. 使用双引号标识字符串

使用双引号标识的字符串中不能包含双引号，具体如下所示：

```
"xiaoqian"
"666"
"I'll do my best."
```

3．使用三引号标识字符串

使用 3 对单引号或 3 对双引号标识字符串可以包含多行，具体如下所示：

```
'''
坚持到感动自己
拼搏到无能为力
'''
"""
遇到 IT 技术难题
就上扣丁学堂
"""
```

这种形式的字符串经常出现在函数定义的下一行，用来说明函数的功能。

通常使用前两种形式创建字符串，之后需要通过变量引用字符串，具体示例如下：

```
name = "小千"
print(name)  # 输出小千
```

注意 Python 中的字符串不能被修改，具体示例如下：

```
name = "xiaoqian"
name[4] = 'f'    # 错误
print(name[4])   # 正确
```

虽然字符串不可以修改，但可以截取字符串一部分与其他字符串进行连接，具体示例如下：

```
str = "xiaoqian is a programmer."
print(str[0:14] + "girl")
```

上述示例中，str[0:14]截取"xiaoqian is a "，然后再与"girl"进行连接，最后输出"xiaoqian is a girl"。字符串的截取与连接将会在后面详细讲解。

4.1.2 转义字符

字符串中除了可以包含数字字符、字母字符或特殊字符外，还可以包含转义字符。转义字符以反斜杠“\”开头，后跟若干个字符。转义字符具有特定的含义，不同于字符原有的意义，故称转义字符。表 4.1 列出了常用的转义字符及含义。

表 4.1 常用的转义字符及含义

转 义 字 符	说 明
\（在行尾时）	续行符
\\	反斜杠符
\n	回车换行
\t	横向制表符
\b	退格
\r	回车
\f	换页
\'	单引号符
\"	双引号符
\a	鸣铃
\ddd	1～3 位八进制数所代表的字符
\xhh	1～2 位十六进制数所代表的字符

在表 4.1 中，'\ddd'和'\xhh'都是用 ASCII 码表示一个字符，如'\101'和'\x41'都是表示字符'A'。转义字符在输出中有许多应用，如想在单引号标识的字符串中包含单引号，则可以使用如下语句：

```
str = 'I\'ll do my best.'
```

其中，“\'”表示对单引号进行转义。当解释器遇到这个转义字符时就理解这不是字符串结束标记。如果想禁用字符串中反斜杠转义功能，可以在字符串前面添加一个 r，具体示例如下：

```
print(r'\n 表示回车换行')  # 输出\n 表示回车换行
```

4.2 字符串的输出与输入

在实际开发中，程序经常需要用户输入字符串并进行处理。字符串被处理完成后，又需要输出显示。上述过程就涉及字符串的输入与输出。

4.2.1 字符串的输出

最简单的字符串输出如下所示：

```
print("xiaoqian") # 输出 xiaoqian
```

此外，Python 支持字符串格式化输出，具体示例如下：

```
age = 18
print("小千的年龄为%d"%age)  # 输出小千的年龄为 18
```

字符串格式化是指按照指定的规则连接、替换字符串并返回新的符合要求的字符串，例如示例中 age 的内容 18 以整数形式替换到要显示的字符串中。字符串格式化的语法格式如下：

```
format_string % string_to_convert
format_string % (string_to_convert1, string_to_convert2, …)
```

其中，format_string 为格式标记字符串，包括固定的内容与待替换的内容，待替换的内容用格式化符号标明，string_to_convert 为需要格式化的数据。如果需要格式化的数据是多个，则需要使用小括号括起来并用逗号分隔。format_string 中常用的格式化符号如表 4.2 所示。

表 4.2　常用格式化符号

格式化符号	说　明
%c	格式化字符
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数（十六进制字母小写）
%X	格式化无符号十六进制数（十六进制字母大写）
%f 或%F	格式化浮点数字，可指定小数点后的精度
%e	用科学记数法格式化浮点数（e 使用小写显示）
%E	用科学记数法格式化浮点数（E 使用大写显示）
%g	由 Python 根据数字的大小自动判断转换为%e 或%f
%G	由 Python 根据数字的大小自动判断转换为%E 或%F
%%	输出%

接下来演示格式化符号的用法，如例 4-1 所示。

例 4-1　格式化符号的用法。

```
1   name, age, id, score = "小千", 18, 1, 95.5
2   print("学号:%d\n 姓名:%s\n 年龄:%d\n 成绩:%f"\
3         %(id, name, age, score))
```

运行结果如图 4.1 所示。

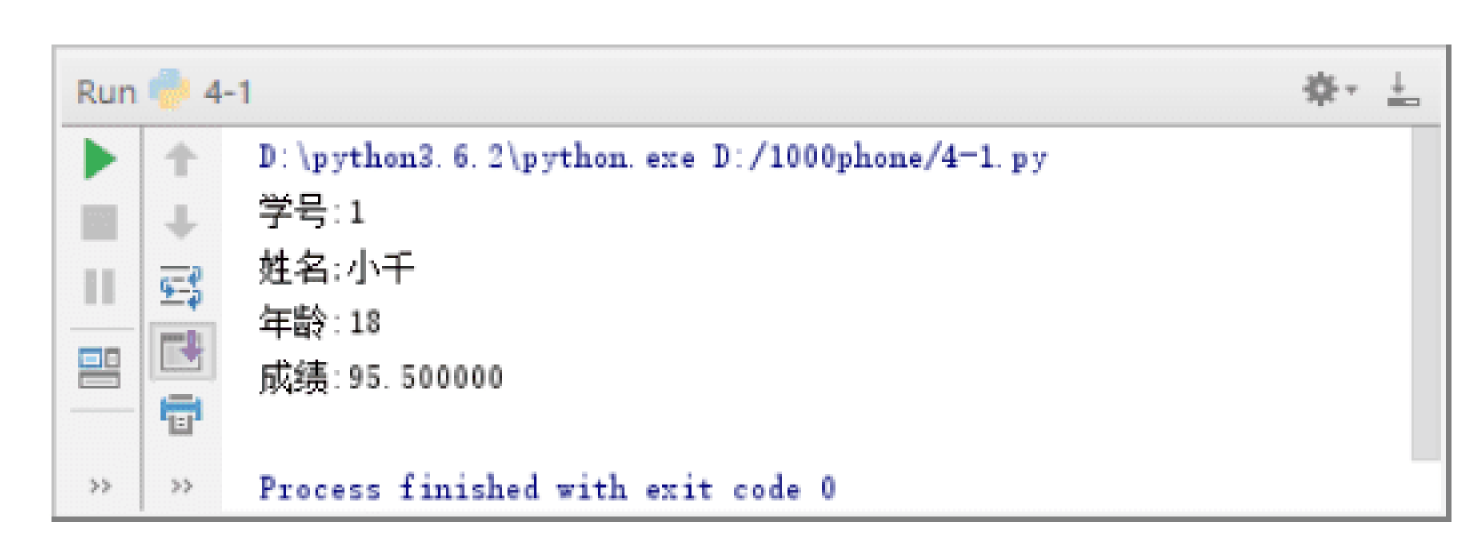


图 4.1　例 4-1 运行结果

在例 4-1 中，第 1 行定义 4 个变量，分别为 name、age、id、score。第 2 行在 print()

函数中格式标记字符串，第 3 行为需要格式化的数据。

除了表 4.2 中的格式化符号，有时还需要调整格式化符号的显示样式，例如是否显示正值符号“+”，表 4.3 中列出了辅助格式化符号。

表 4.3 辅助格式化符号

辅助格式化符号	说　明
*	定义宽度或小数点的精度
-	左对齐
+	对正数输出正值符号“+”
#	在八进制数前显示 0，在十六进制前显示 0x 或 0X
m.n	m 是显示的最大总宽度，n 是小数点后的位数
<sp>	数字的大小不满足 m.n 时，用空格补位
0	数字的大小不满足 m.n 时，用 0 补位

接下来演示辅助格式化符号的用法，如例 4-2 所示。

例 4-2 辅助格式化符号的用法。

```
1   a, b = 65, 3.1415926
2   print("%#10x"%a)
3   print("%-#10X"%a)
4   print("%+d"%a)
5   print("%5.3f"%b)
6   print("%*.3f"%(5, b))
7   print("%5.*f"%(3, b))
```

运行结果如图 4.2 所示。

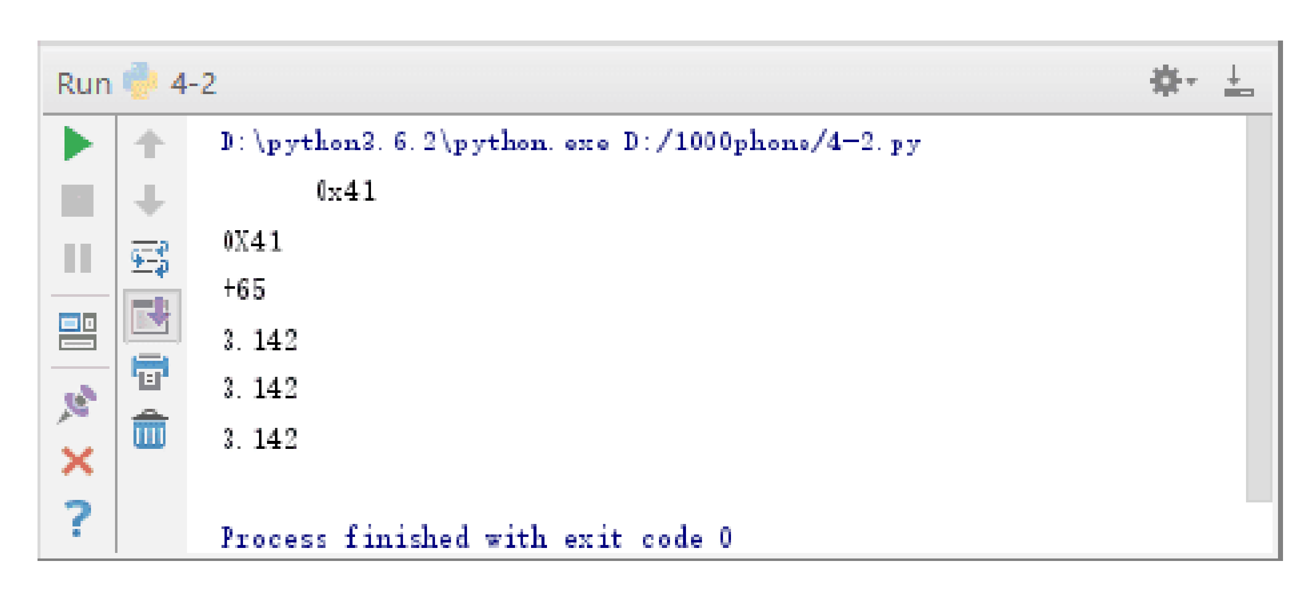


图 4.2 例 4.2 运行结果

在例 4-2 中，第 2 行输出字符串宽度为 10，并且以 0x 形式显示 65 对应的十六进制数，注意默认是右对齐的。第 3 行输出字符串宽度为 10，并且以 0X 形式显示 65 对应的十六进制数，注意“-”代表左对齐。第 4 行输出字符串中正值时前加“+”。第 5 行输出字符串宽度为 5，显示的小数点精度为 3。第 6 行通过*设置显示宽度为 5。第 7 行通过*设置小数点精度为 3。

4.2.2 字符串的输入

在前面的程序中，字符串都是先定义后使用。如果需在程序运行时，通过键盘输入

字符串，则可以使用 input()函数。它表示从标准输入读取一行文本，默认的标准设备是键盘，其语法格式如下：

```
input([prompt])
```

其中，prompt 表示提示字符串，该函数将输入的数据作为字符串返回。

接下来演示其用法，如例 4-3 所示。

例 4-3 input()函数的用法。

```
1  name = input("请输入用户名：")
2  pwd = input("请输入密码：")
3  print("用户%s 的密码为%s"%(name, pwd))
4  print(type(name))
5  print(type(pwd))
```

运行结果如图 4.3 所示。

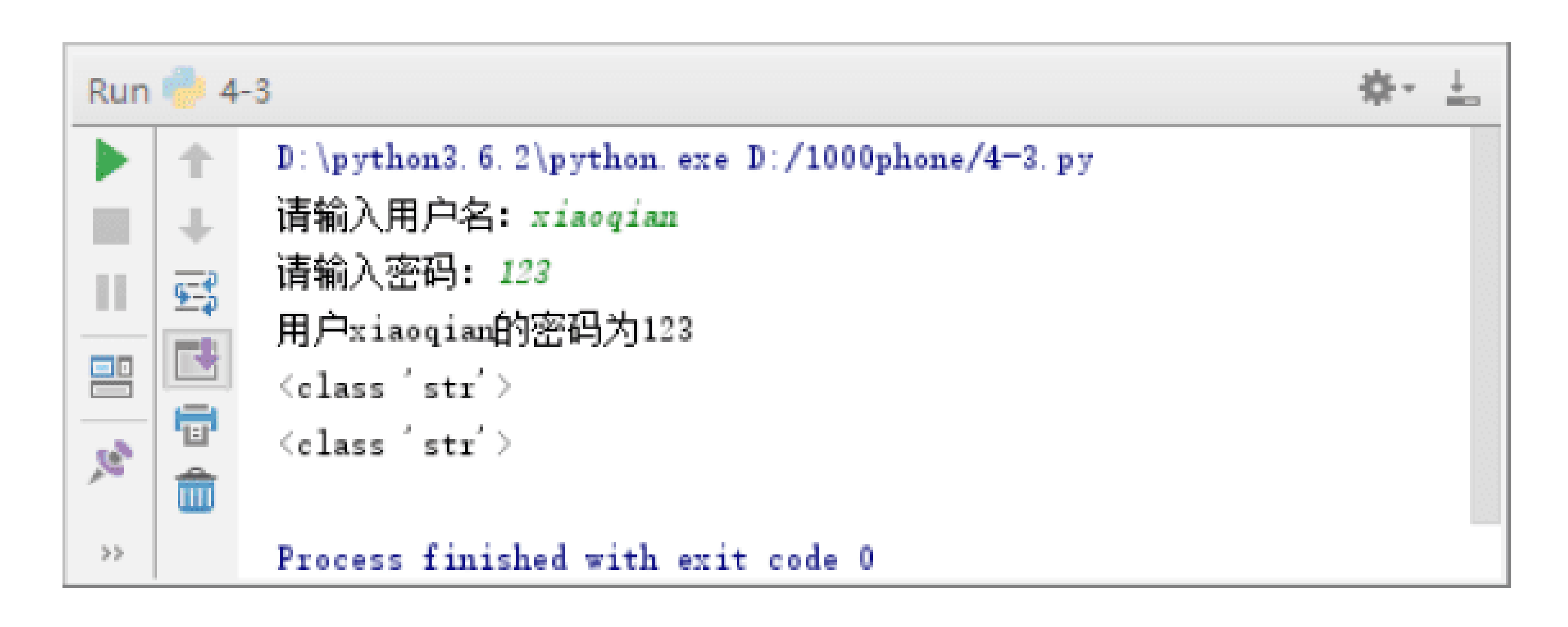


图 4.3 例 4-3 运行结果

在例 4-3 中，第 1 行与第 2 行分别通过 input()函数从键盘输入字符串并通过变量名引用相应的字符串。第 3 行输出字符串。从运行结果可以看出，当从键盘输入 123 时，pwd 最终的数据类型为字符串类型。

4.3 字符串的索引与切片

字符串可以通过运算符[]进行索引与切片，字符串中每个字符都对应两个编号（也称下标），如图 4.4 所示。

str	w	w	w	.	q	f	e	d	u	.	c	o	m
index	0	1	2	3	4	5	6	7	8	9	10	11	12
	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

图 4.4 字符串下标

在图 4.4 中，字符串 str 正向编号从 0 开始，代表第一个字符，依次往后；字符串 str 负向编号从−1 开始，代表最后一个字符，依次往前。因为编号可正可负，所以字符串中的某个字符可以有两种方法索引，例如索引 str 中字符'q'，具体示例如下：

```
str[4]
str[-9]
```

上述两种形式都可以索引到字符'q'。

字符串切片是指从字符串中截取部分字符并组成新的字符串，并不会对原字符串做任何改动，其语法格式如下：

```
str[起始编号:结束编号:步长]
```

该语句表示从起始编号处开始，以指定步长进行截取，到结束编号的前一位结束。

接下来演示字符串的切片，如例 4-4 所示。

例 4-4 字符串的切片。

```
str = "www.qfedu.com"
print(str[4:9])       #  默认步长为 1
print(str[4:9:2])     #  设置步长为 2
print(str[-9:-4])     #  默认步长为 1
print(str[-9:-4:2])   #  设置步长为 2
print(str[:])         #  整个字符串
print(str[:9])        #  等价于 str[0:9:1]
print(str[4:])        #  默认到字符串尾部（包括最后一个字符）
print(str[:-9])       #  从第一个字符到编号为-9 的字符（不包括编号为-9 的字符）
print(str[-4:])       #  从编号为-4 的字符到最后一个字符（包括最后一个字符）
print(str[::-2])      #  从后往前，步长为 2
```

运行结果如图 4.5 所示。

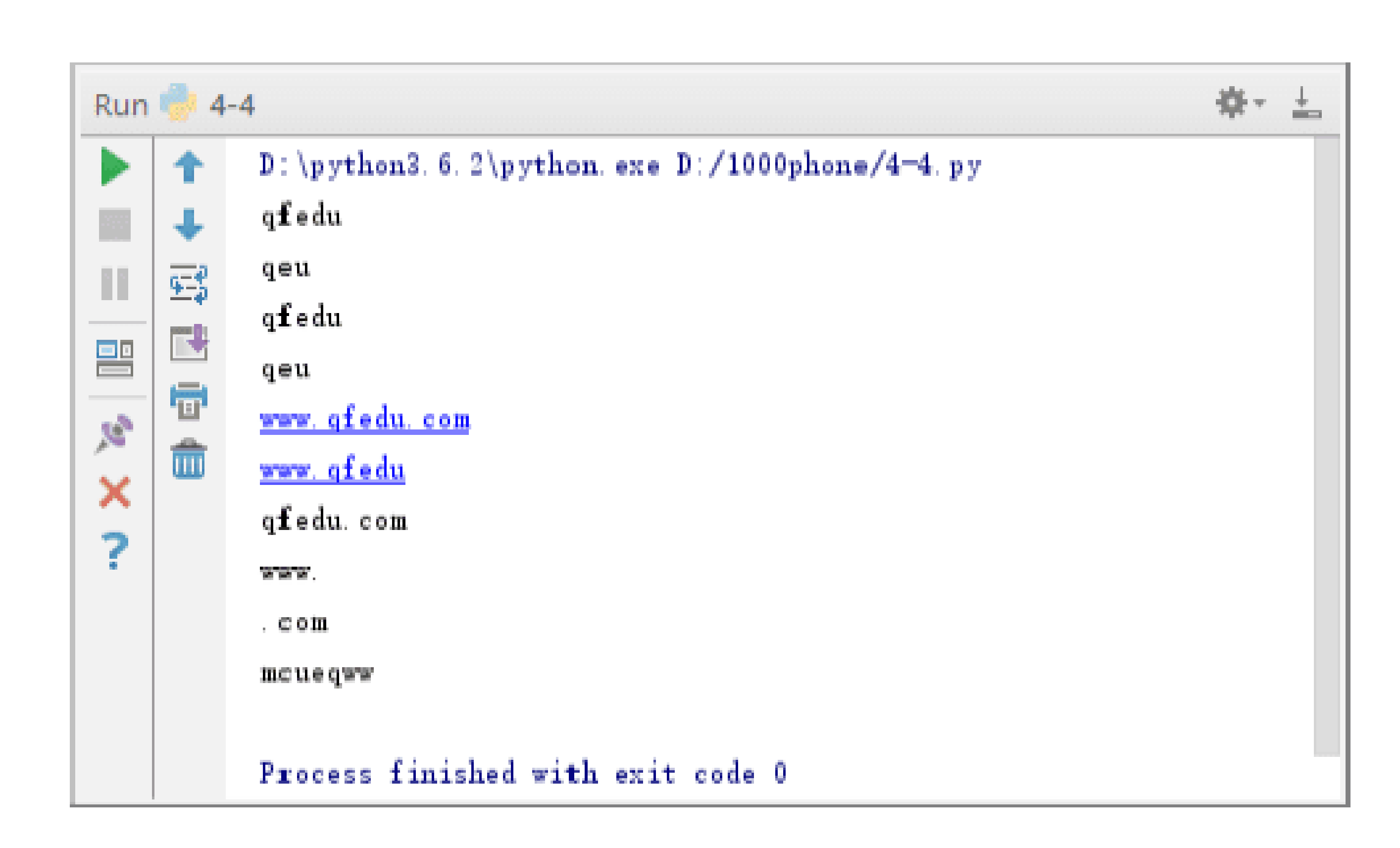


图 4.5 例 4-4 运行结果

在例 4-4 中，每种形式中第一个冒号两边表示切片从何处开始及到何处结束，第二个冒号后表示步长。

4.4 字符串的运算

除了数字类型的数据可以参与运算外，字符串也可以参与运算，如 4.3 节中字符串

通过[]运算符进行索引与切片，具体如表 4.4 所示。

表 4.4　字符串运算

运　算　符	说　　明
+	字符串连接
*	重复字符串
[]	索引字符串中的字符
[:]	对字符串进行切片
in	如果字符串中包含给定字符，返回 True
not in	如果字符串中包含给定字符，返回 False
r 或 R	原样使用字符串

接下来演示字符串的运算，如例 4-5 所示。

例 4-5　字符串的运算。

```
1  str1, str2 = "扣丁", "学堂"
2  print(str1 + str2)
3  print(3 * (str1 + str2))
4  if "coding" in "coding.com":
5      print("coding is in coding.com")
6  else:
7      print("coding is not in coding.com")
```

运行结果如图 4.6 所示。



图 4.6　例 4-5 运行结果

在例 4-5 中，第 2 行输出两个字符串连接的结果，第 3 行输出两个字符串连接并重复 3 次的结果，第 4 行判断字符串"coding.com"中是否包含"coding"。

4.5　字符串常用函数

在程序开发中，字符串经常需要被处理，例如，求字符串的长度、大小写转换等。如果每次处理字符串时，都编写相应的代码，那么开发效率会非常低下，为此 Python 提供了一些内置函数用于处理字符串常见的操作。

4.5.1　大小写转换

Python 中涉及字符串大小写转换的函数，如表 4.5 所示。

表 4.5　大小写转换函数

函　　数	说　　明
upper()	将字符串中所有小写字母转换为大写
lower()	将字符串中所有大写字母转换为小写

上述两种方法都返回一个新字符串，其中的非字母字符保持不变。如果需要进行大小写无关的比较，则这两个函数非常有用。接下来演示其用法，如例 4-6 所示。

例 4-6　lower()函数的用法。

```
1    name = "xiaoqian"  # 假设用户名为 xiaoqian
2    str = input("请输入用户名（不区分大小写）：")
3    if str.lower() == name:
4        print("欢迎用户%s 登录"%name)
5    else:
6        print("用户名错误")
```

运行结果如图 4.7 所示。

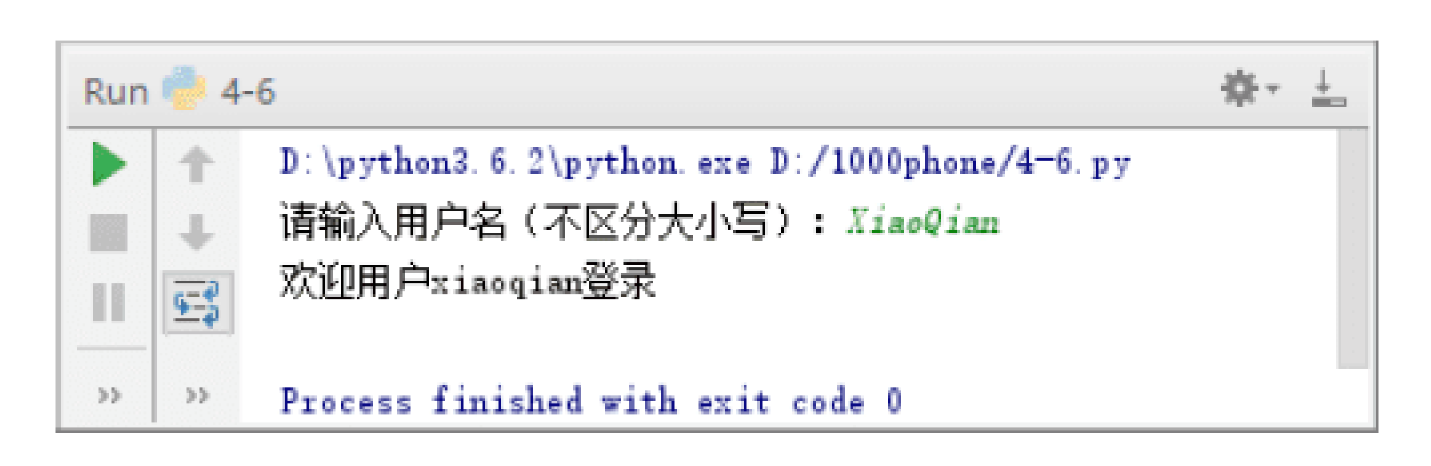


图 4.7　例 4-6 运行结果

在例 4-6 中，当程序运行时，用户通过键盘输入 XiaoQian。第 3 行将字符串 str 通过 lower()函数转换为小写并与 name 进行比较，如果相等，则登录成功，否则登录失败。

4.5.2　判断字符

Python 中提供了判断字符串中包含某些字符的函数，这些函数在处理用户输入的字符串时是非常方便的。这些函数都是以 is 开头，如表 4.6 所示。

表 4.6　判断字符函数

函　　数	说　　明
isupper()	如果字符串中包含至少一个区分大小写的字符，并且所有这些（区分大小写）字符都是大写，则返回 True，否则返回 False
islower()	如果字符串中包含至少一个区分大小写的字符，并且所有这些（区分大小写）字符都是小写，则返回 True，否则返回 False
isalpha()	如果字符串至少有一个字符并且所有字符都是字母，则返回 True，否则返回 False
isalnum()	如果字符串至少有一个字符并且所有字符都是字母或数字，则返回 True,否则返回 False
isdigit()	如果字符串只包含数字，则返回 True，否则返回 False
isspace()	如果字符串中只包含空白，则返回 True，否则返回 False
istitle()	如果字符串是标题化的，则返回 True，否则返回 False

接下来演示这些函数的基本用法，如例 4-7 所示。

例 4-7 判断字符函数的用法。

```
print("xiaoqian".islower())          # True
print("Xiaoqian".islower())          # 小写字母中有大写字母
print("xiaoqian6666".islower())      # True
print("XIAOQIAN".isupper())          # True
print("XIAOqIAN".isupper())          # 大写字母中有小写字母
print("XIAOQIAN6666".isupper())      # True
print("xiaoqian666".isalpha())       # 包含数字字符
print("xiaoqian666".isalnum())       # True
print("xianqian666".isdigit())       # 包含字母字符
print(" \t\n".isspace())             # True
print("Title".istitle())             # True
```

运行结果如图 4.8 所示。

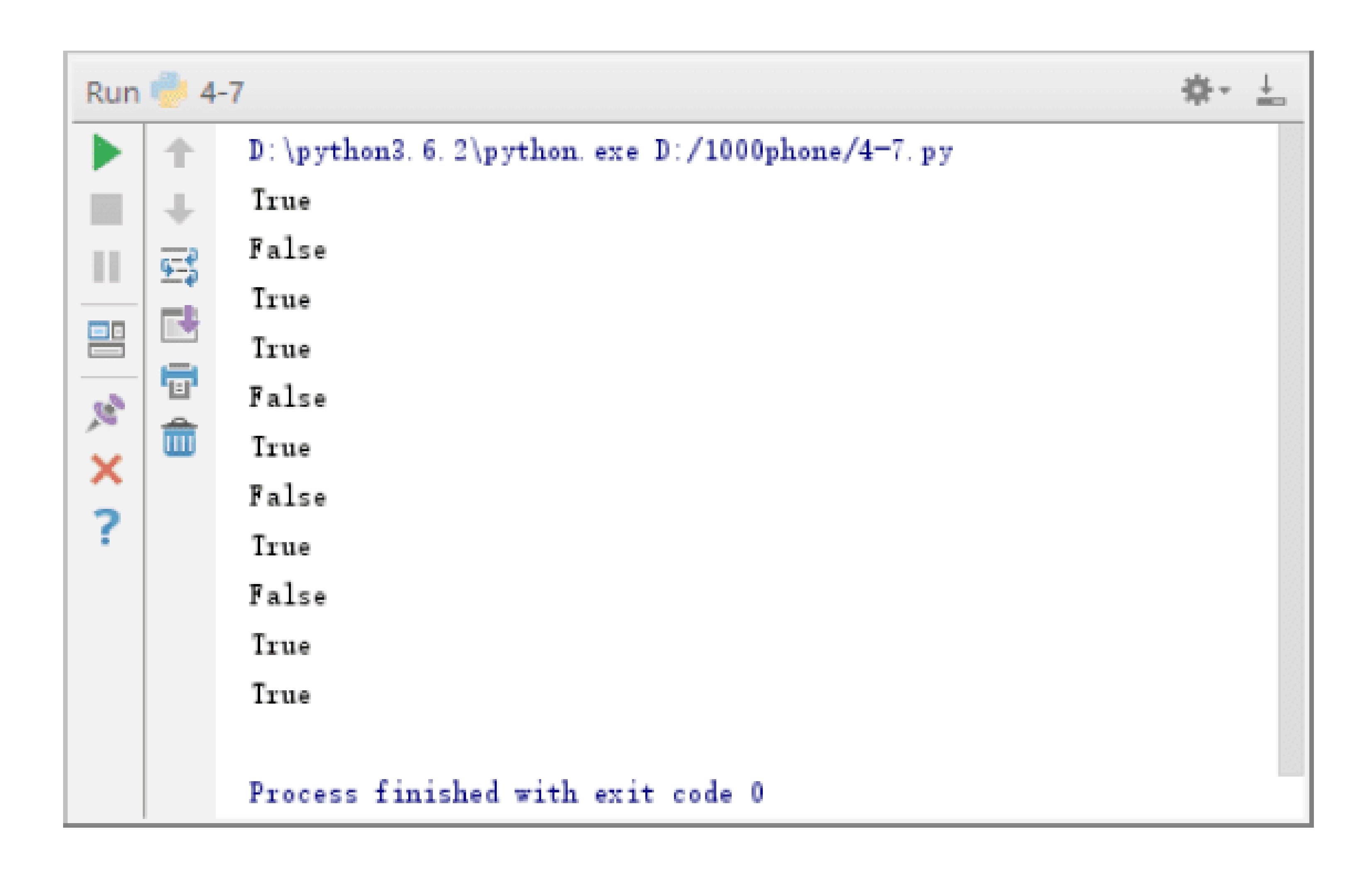


图 4.8 例 4-7 运行结果

在例 4-7 中，这些函数的返回值都为布尔值。接下来演示使用这些函数验证用户输入的密码是否符合格式要求，如例 4-8 所示。

例 4-8 验证密码是否符合格式要求。

```
while True:
    pwd = input("请输入您的密码（必须包含数字与字母）：")
    if pwd.isalnum() and (not pwd.isalpha()) and (not pwd.isdigit()):
        print("您的密码为%s"%pwd)
        break
    else:
        print("重新输入！")
```

运行结果如图 4.9 所示。

在例 4-8 中，程序通过循环判断用户输入的密码，其中必须包含数字与字母。从程序运行结果可看出，当输入的密码包含数字和字母时，程序才会退出循环；否则，一直

提示用户输入密码。

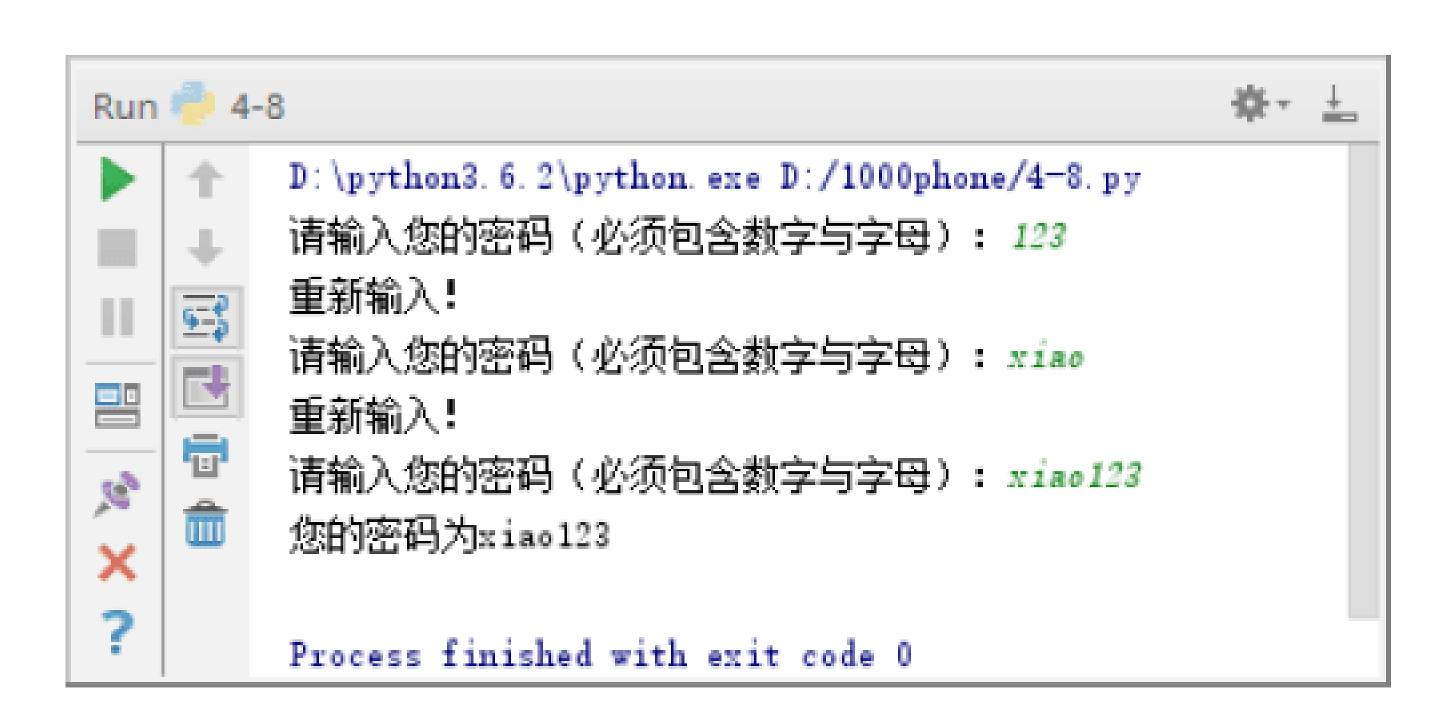


图 4.9　例 4-8 运行结果

4.5.3　检测前缀或后缀

在处理字符串时，有时需要检测字符串是否以某个前缀开头或以某个后缀结束，这时可以使用 startswith()与 endswith()函数，如表 4.7 所示。

表 4.7　检测前缀或后缀函数

函　数	说　明
startswith(prefix, beg=0,end=len(string))	检查字符串是否是以 prefix 开头，如果是，则返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查
endswith(suffix, beg=0, end=len(string))	检查字符串是否以 suffix 结束，如果是，返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查

接下来演示这两个函数的用法，如例 4-9 所示。

例 4-9　startswith()与 endswith()函数的用法。

```
str = "www.codingke.com"
print(str.startswith("www"))
print(str.startswith("coding", 4))
print(str.endswith("com"))
print(str.endswith("ke", 0, 12))
```

运行结果如图 4.10 所示。

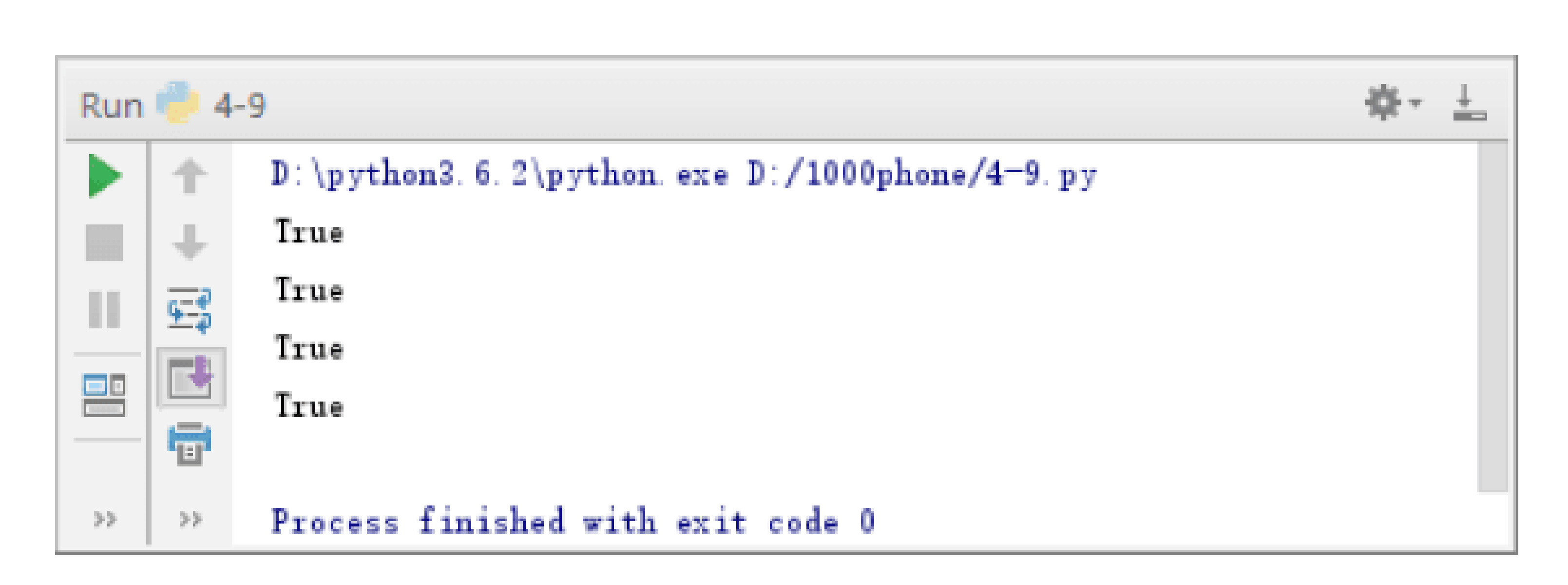


图 4.10　例 4-9 运行结果

在例 4-9 中，startswith()与 endswith()函数中后两个参数代表检测字符串的范围。这

两个函数用于检测字符串开始或结束的部分是否等于另一个字符串，其作用与等于操作符类似，这使编程更加灵活。

4.5.4 合并与分隔字符串

在处理字符串时，有时需要合并与分隔字符串，这时可以使用 join()与 split()函数，如表 4.8 所示。

表 4.8 合并与分隔函数

函　数	说　明
join(seq)	以指定字符串作为分隔符，将 seq 中所有的元素（字符串表示）合并为一个新的字符串
split(str="", num=string.count(str))	以 str 为分隔符分隔字符串，如果 num 有指定值，则仅分隔 num 次

接下来演示这两个函数的用法，如例 4-10 所示。

例 4-10 join()与 split()函数的用法。

```
1  seq1 = "千锋教育"  # 字符串
2  print("|".join(seq1))
3  seq2 = ["千锋教育", "扣丁学堂", "好程序员特训营"] # 列表
4  print("-".join(seq2))
5  str3 = "千锋教育|扣丁学堂|好程序员特训营"
6  print(str3.split("|"))
7  print(str3.split("|", 1))
```

运行结果如图 4.11 所示。

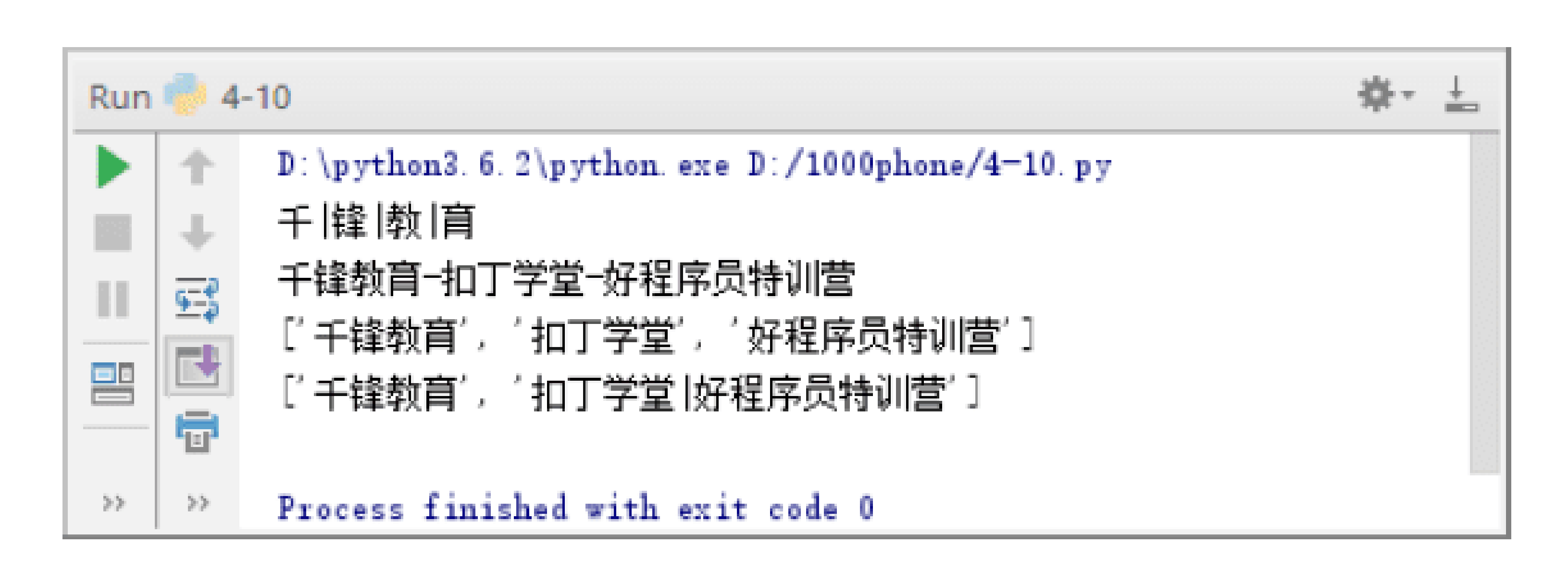


图 4.11 例 4-10 运行结果

在例 4-10 中，第 2 行将“|”与字符串 seq1 中的每个字符合并成一个新字符串。第 4 行将“-”与列表 seq2 中的每个元素合并成一个新字符串。第 6 行将字符串 str3 以“|”为分隔符进行分隔。第 7 行指定分隔次数为 1。

4.5.5 对齐方式

在处理字符串时，有时需要设置字符串对齐方式，这时可以使用 rjust()、ljust()和

center()函数，如表 4.9 所示。

表 4.9　对齐方式函数

函　　数	说　　明
rjust(width,[, fillchar])	返回一个原字符串右对齐，并使用 fillchar（默认空格）填充至长度 width 的新字符串
ljust(width[, fillchar])	返回一个原字符串左对齐，并使用 fillchar（默认空格）填充至长度 width 的新字符串
center(width, fillchar)	返回一个原字符串居中，并使用 fillchar（默认空格）填充至长度 width 的新字符串

接下来演示这 3 个函数的用法，如例 4-11 所示。

例 4-11　对齐方式函数的用法。

```
1   str = "千锋教育"    # 字符串
2   print(str.rjust(10))
3   print(str.rjust(10, '$'))
4   print(str.ljust(10))
5   print(str.ljust(10, '$'))
6   print(str.center(10))
7   print(str.center(10, '$'))
```

运行结果如图 4.12 所示。

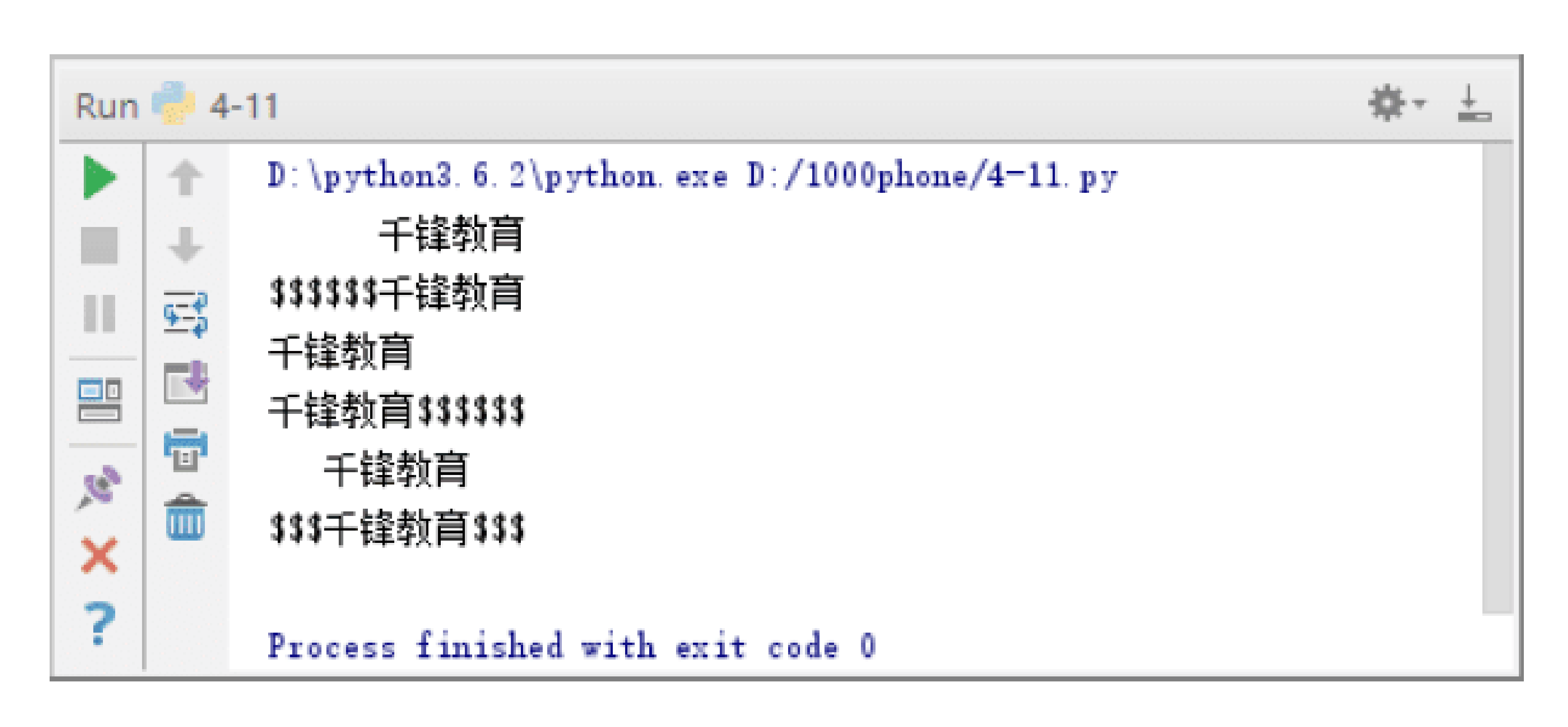


图 4.12　例 4-11 运行结果

在例 4-11 中，第 2 行设置宽度为 10、右对齐、空格填充方式显示新字符串。第 3 行设置宽度为 10、右对齐、$填充方式显示新字符串。后面的几个函数与前面函数的用法类似，在此不再赘述。

4.5.6　删除字符串头尾字符

在处理字符串时，有时需要删除字符串头尾的某些字符，这时可以使用 strip()、lstrip()和 rstrip()函数，如表 4.10 所示。

表 4.10　删除字符串头尾字符函数

函　　数	说　　明
strip([chars])	删除字符串头尾指定的 chars 字符，默认删除空白字符
lstrip([chars])	删除字符串头部指定的 chars 字符，默认删除空白字符
rstrip()	删除字符串尾部指定的 chars 字符，默认删除空白字符

接下来演示这 3 个函数的用法，如例 4-12 所示。

例 4-12　删除字符串头尾字符函数的用法。

```
1    str1, str2, str3 = "\t 千锋教育\t", "***扣丁学堂***", "goodprogrammer"
2    print(str1.strip())
3    print(str2.strip('*'))
4    print(str3.strip('good'))
5    print(str3.strip('odg'))
6    print(str2.lstrip('*'))
7    print(str2.rstrip('*'))
```

运行结果如图 4.13 所示。

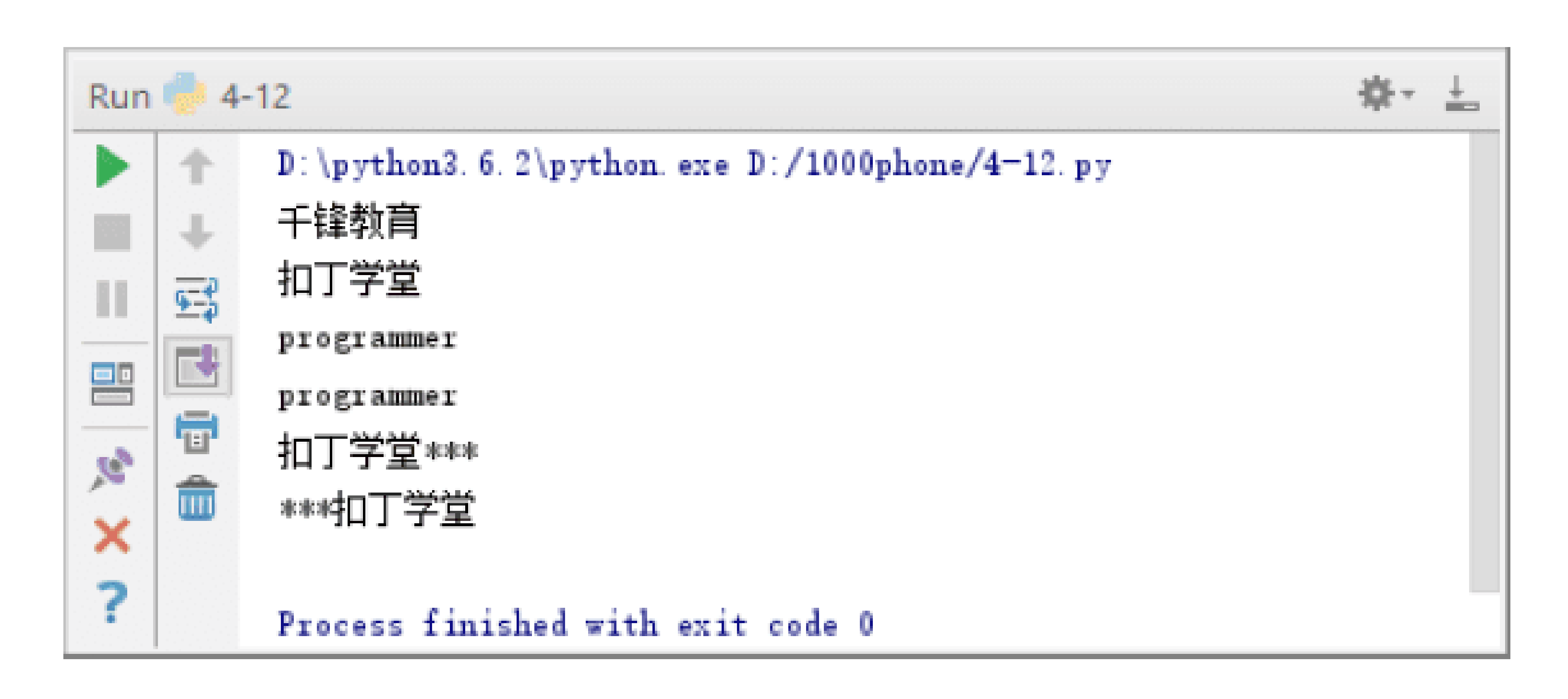


图 4.13　例 4-12 运行结果

在例 4-12 中，第 4 行与第 5 行输出的结果相同，说明在指定删除字符时，字符的顺序并不重要，只需保证包含的字符相同，便可得到想要的结果。

4.5.7　检测子串

在处理字符串时，有时需要检测字符串中是否包含某个子字符串，这时可以使用 find()函数，其语法格式如下：

```
find(str, beg = 0, end = len(string))
```

该函数检测 str 是否包含在检测字符串中。如果指定范围 beg 和 end，则检查是否包含在指定范围内。如果包含，则返回开始字符的下标值，否则返回−1。

接下来演示该函数的用法，如例 4-13 所示。

例 4-13　find()函数的用法。

```
str = "遇到 IT 技术难题,就上扣丁学堂"
```

```
2    print(str.find("IT"))
3    print(str.find("Python"))
4    print(str.find("扣丁学堂", 10))
5    print(str.find("扣丁学堂", 10, 14))
```

运行结果如图 4.14 所示。

```
Run 4-13
D:\python3.6.2\python.exe D:/1000phone/4-13.py
2
-1
11
-1

Process finished with exit code 0
```

图 4.14　例 4-13 运行结果

在例 4-13 中，第 2 行在字符串 str 中查找是否包含 IT，结果返回 IT 在字符串中的下标。第 3 行在字符串 str 中查找是否包含 Python，结果返回−1，说明不包含。第 4 行指定查找范围从下标为 10 的字符开始到字符串结尾（包含最后一个字符）。第 5 行指定查找范围从下标为 10 的字符开始到下标为 14 的前一个字符结束。

除此之外，还可以通过 index()函数检测字串，其语法格式如下：

```
index(str, beg=0, end=len(string))
```

该函数的用法与 find()函数类似，两者的区别是：如果 str 不在字符串中，那么 index()函数会报一个异常。

接下来演示其用法，如例 4-14 所示。

例 4-14　index()函数的用法。

```
1    str = "遇到 IT 技术难题,就上扣丁学堂"
2    print(str.index("扣丁学堂", 10))
3    print(str.index("扣丁学堂", 10, 14))
```

运行结果如图 4.15 所示。

```
Run 4-14
D:\python3.6.2\python.exe D:/1000phone/4-14.py
11
Traceback (most recent call last):
  File "D:/1000phone/4-14.py", line 3, in <module>
    print(str.index("扣丁学堂", 10, 14))
ValueError: substring not found

Process finished with exit code 1
```

图 4.15　例 4-14 运行结果

在例 4-14 中，第 3 行在字符串 str 中没有检测到"扣丁学堂"，此时会抛出一个异常，如图 4.15 中显示的"ValueError: substring not found"。

4.5.8 替换子串

在文字处理软件中，都会有查找并替换的功能。在字符串中，可以通过 replace()函数来实现，其语法格式如下：

```
replace(old, new [, max])
```

该函数将字符串中 old 替换成 new 并返回新生成的字符串。如果指定第三个参数 max，则表示替换不超过 max 次。

接下来演示该函数的用法，如例 4-15 所示。

例 4-15 replace()函数的用法。

```
1   str = "Anything I do, I spend a lot of time."
2   print(str.replace('I', 'you'))
3   print(str.replace('I', 'you', 1))
```

运行结果如图 4.16 所示。

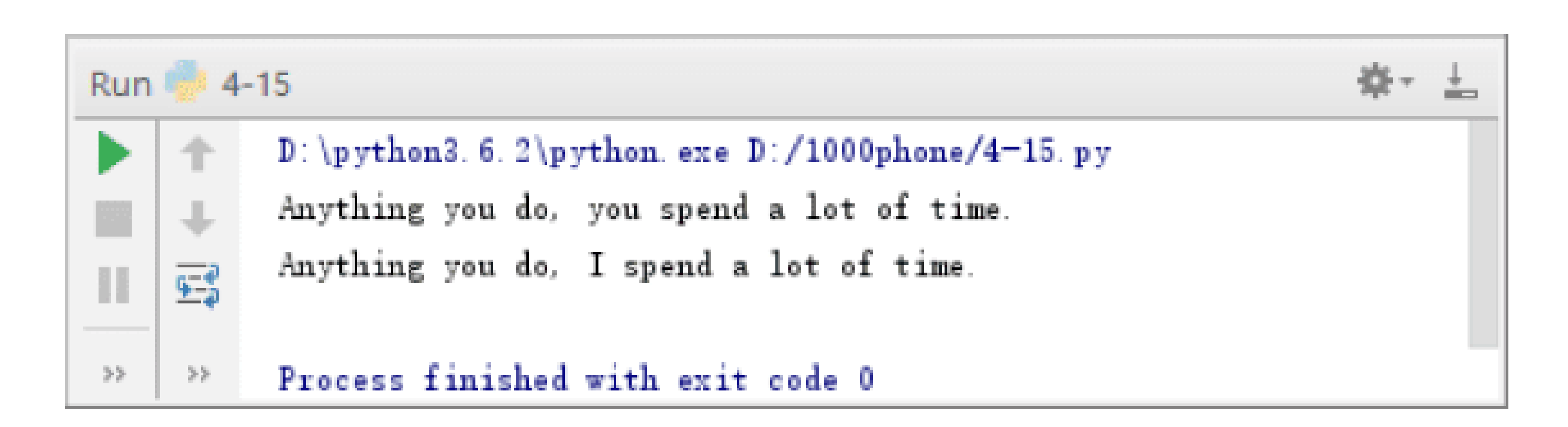


图 4.16 例 4-15 运行结果

在例 4-15 中，第 2 行将字符串 str 中所有的字符'I'替换为'you'，第 3 行将字符串 str 中的字符'I'只替换一次为'you'。

4.5.9 统计子串个数

在文字处理软件中，都会有统计某个词语出现次数的功能。在字符串中，可以通过 count()函数来实现，其语法格式如下：

```
count(str, beg = 0, end = len(string))
```

该函数返回 str 在字符串中出现的次数。如果指定 beg 或 end，则返回指定范围内 str 出现的次数。

接下来演示该函数的用法，如例 4-16 所示。

例 4-16 count()函数的用法。

```
str = "Anything I do, I spend a lot of time."
print(str.count('I'))
print(str.count('I', 0, 10))
```

运行结果如图 4.17 所示。

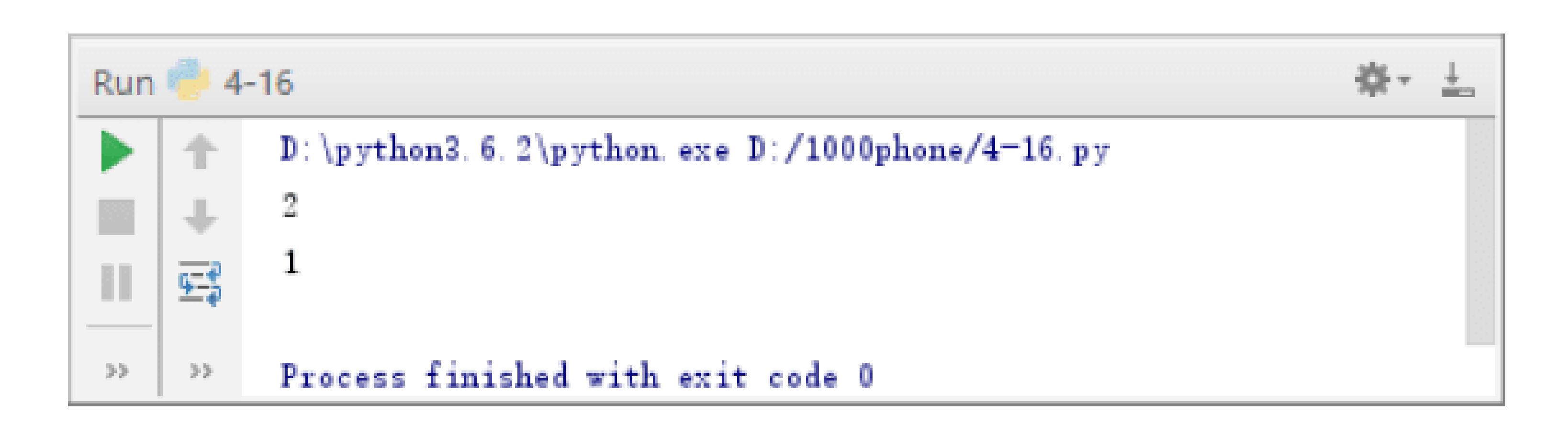


图 4.17　例 4-16 运行结果

在例 4-16 中，第 2 行统计字符串 str 中字符'I'出现的次数，第 3 行统计在下标为 0 到下标为 10 的前一位字符之间'I'出现的次数。

4.5.10　首字母大写

capitalize()函数用于将字符串的第一个字母变成大写，其他字母变成小写，其语法格式如下：

```
capitalize()
```

接下来演示该函数的用法，如例 4-17 所示。

例 4-17　capitalize()函数的用法。

```
str = "Codingke is a great website."
print(str.capitalize())
```

运行结果如图 4.18 所示。

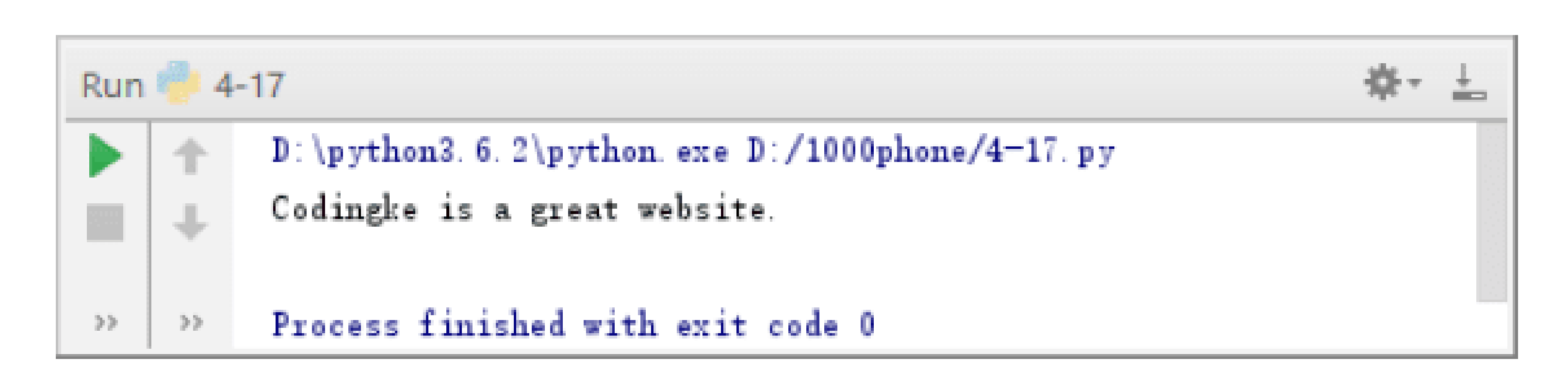


图 4.18　例 4-17 运行结果

从运行结果可看出，输出的字符串首字母变成大写并且其他字母变成小写。

4.5.11　标题化

title()函数可以将字符串中所有单词首字母大写，其他字母小写，从而形成标题，其语法格式如下：

```
title()
```

接下来演示该函数的用法，如例 4-18 所示。

例 4-18　title()函数的用法。

```
str = "Qianfeng education"
```

```
print(str.title())
```

运行结果如图 4.19 所示。

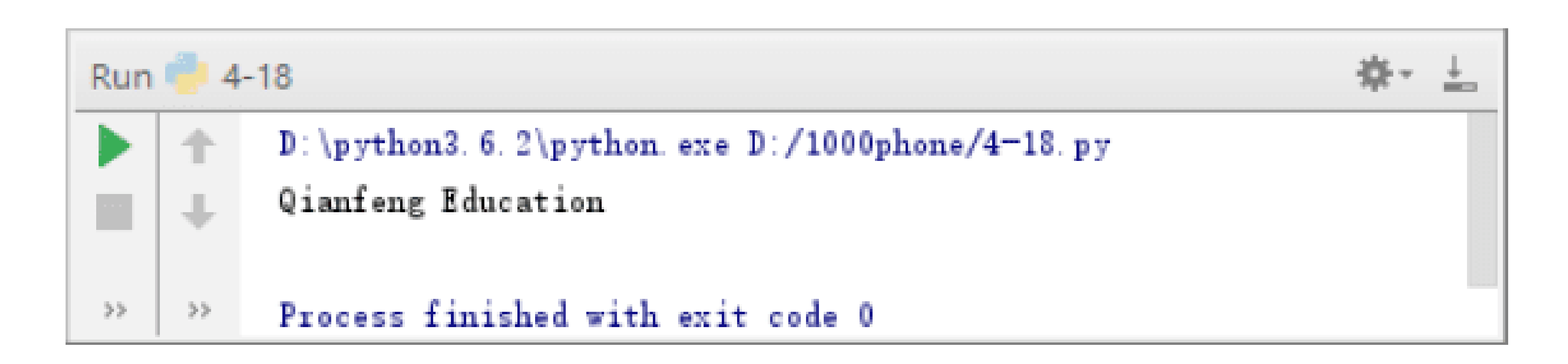


图 4.19　例 4-18 运行结果

从运行结果可看出，输出的字符串中每个单词首字母变成大写并且其他字母变成小写。

4.6 小　案　例

在注册网站时，用户经常需要设置密码，然后程序根据用户输入的密码判断安全级别，具体如下所示：

- 低级密码——包含单纯的数字或字母，长度小于等于 8。
- 中级密码——必须包含数字、字母或特殊字符（仅限：~!@#$%^&*()_=-/,.?<>;:[]{}|\）中的任意两种，长度大于 8。
- 高级密码——必须包含数字、字母及特殊字符（仅限：~!@#$%^&*()_=-/,.?<>;:[]{}|\）中的 3 种，长度大于 16。

接下来按照上述要求编写程序，如例 4-19 所示。

例 4-19　根据用户输入的密码判断安全级别。

```
# 字符分类,number 保存数字字符,letter 保存字母字符,symbols 保存其他字符
number = '0123456789'
letter = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
symbols = r'''`!@#$%^&*()_+-=/*{}[]\|'";:/?,.<>'''
# 输入密码
pwd = input('请输入密码：')
# 判断长度
length = len(pwd)
# 如果输入的密码为空或空格,重新输入
while (pwd.isspace() or length == 0) :
    pwd = input("您输入的密码为空（或空格），请重新输入：")
# 判断长度等级
if length <= 8:
    lenGrade = 1
elif 8 < length < 16:
    lenGrade = 2
else:
    lenGrade = 3
```

```
19 # 标记字符等级
20 charGrade = 0
21 # 判断是否包含数字
22 for each in pwd:
23     if each in number:
24         charGrade += 1
25         break
26 # 判断是否包含字母
27 for each in pwd:
28     if each in letter:
29         charGrade += 1
30         break
31 # 判断是否包含特殊字符
32 for each in pwd:
33     if each in symbols:
34         charGrade += 1
35         break
36 # 判断并打印结果
37 if lenGrade == 1 or charGrade == 1 :
38     print("您的密码安全级别为：低")
39 elif lenGrade == 2 or charGrade == 2 :
40     print("您的密码安全级别为：中")
41 else :
42     print("您的密码安全级别为：高")
```

运行结果如图 4.20 所示。



图 4.20　例 4-19 运行结果

在例 4-19 中，第 2～4 行定义了 3 个变量，分别存储了数字、字母和其他字符；第 13～18 行判断用户输入的字符长度并进行等级划分；第 22～35 行判断用户输入的是否包含数字、字母、其他字符并进行等级划分；第 37～42 行判断安全等级并输出结果。从图 4.20 中可看到输入的包含数字、字母及其他字符且长度大于 16，因此密码安全级别为高。

4.7　本章小结

本章主要介绍了 Python 中的字符串，首先讲解了字符串有 3 种表示方法及字符串中的转义字符，接着讲解了字符串的输入与输出以便与程序更好地交互，又讲解了字符串

的索引与切片，最后讲解了字符串的运算及常用函数。通过本章的学习，应能熟练使用字符串的切片及常用函数。

4.8 习　题

1．填空题

（1）转义字符以________开头。
（2）对字符串进行输出可以使用________函数。
（3）对字符串进行输入可以使用________函数。
（4）删除字符串头尾指定字符的函数是________。
（5）________运算符可以将两个字符串连接起来。

2．选择题

（1）下列不属于字符串的是（　　）。
A．qianfeng　　B．'qianfeng'　　C．"qianfeng"　　D．"""qianfeng"""
（2）使用（　　）符号可以对字符串类型的数据进行格式化。
A．%d　　B．%f　　C．%e　　D．%s
（3）下列函数可以返回某个子串在字符串中出现次数的是（　　）。
A．index()　　B．count()　　C．find()　　D．replace()
（4）若函数 find()没有在字符串中找到子串，则返回（　　）。
A．原字符串　　B．一个异常　　C．0　　D．−1
（5）若 str = "qianfeng"，则 print(str[3:7])输出（　　）。
A．nfen　　B．nfeng　　C．anfen　　D．anfeng

3．思考题



（1）简述字符串的 3 种表现形式。
（2）简述字符串的切片。

4．编程题

输入一个字符串，分别统计出其中字母、数字和其他字符的个数。

第 5 章 chapter 5

列表与元组

本章学习目标

- 掌握列表的概念。
- 掌握列表的常用操作。
- 掌握列表解析。
- 掌握元组的概念。
- 掌握元组的操作。

第 4 章讲解的字符串是简单序列，即字符串中每个元素都是字符。除此之外，列表与元组也是序列，它们的元素可以是不同类型的数据，这使得程序处理不同类型的数据变得更加容易。

5.1 列表的概念

列表是 Python 以及其他语言中最常用到的数据类型之一。Python 中使用中括号[]来创建列表，具体示例如下：

```
student = [20190101, "小千", 18, 99.5]
```

5.1.1 列表的创建

列表是由一组任意类型的值组合而成的序列，组成列表的值称为元素，每个元素之间用逗号隔开，具体示例如下：

```
list1 = [1, 2, 3, 4, 5]                               # 元素为 int 型
list2 = ['千锋教育', '扣丁学堂', '好程序特训营']     # 元素为 String 型
list3 = ['小千', 18, 98.5]                            # 元素为混合类型
list4 = ['千锋教育', ['小千', 18, 98.5]]              # 列表嵌套列表
```

上述示例中，创建了 4 个列表，其中 list4 中嵌套了一个列表，正是由于列表中元素可以是任意类型数据，才使得数据表示更加简单。

此外，还可以创建一个空列表，具体示例如下：

```
list5 = []
```

大家可能会疑惑：创建一个空列表有什么作用？在实际开发中，可能无法提前预知列表中包含多少个元素及每个元素的值，只知道将会用一个列表来保存这些元素。当有了空列表后，程序就可以向这个列表中添加元素。此处需注意，列表中的元素是可变的，这意味着可以向列表中添加、修改和删除元素，如例 5-1 所示。

例 5-1 列表的简单使用。

```
1   name, age, score = '小千', 18, 95.5
2   list1 = [name, age, score]
3   print(list1)
4   name, age, score = '小锋', 20, 100
5   print(list1)
6   print(name, age, score)
7   list1[0] = name
8       print(list1)
```

运行结果如图 5.1 所示。

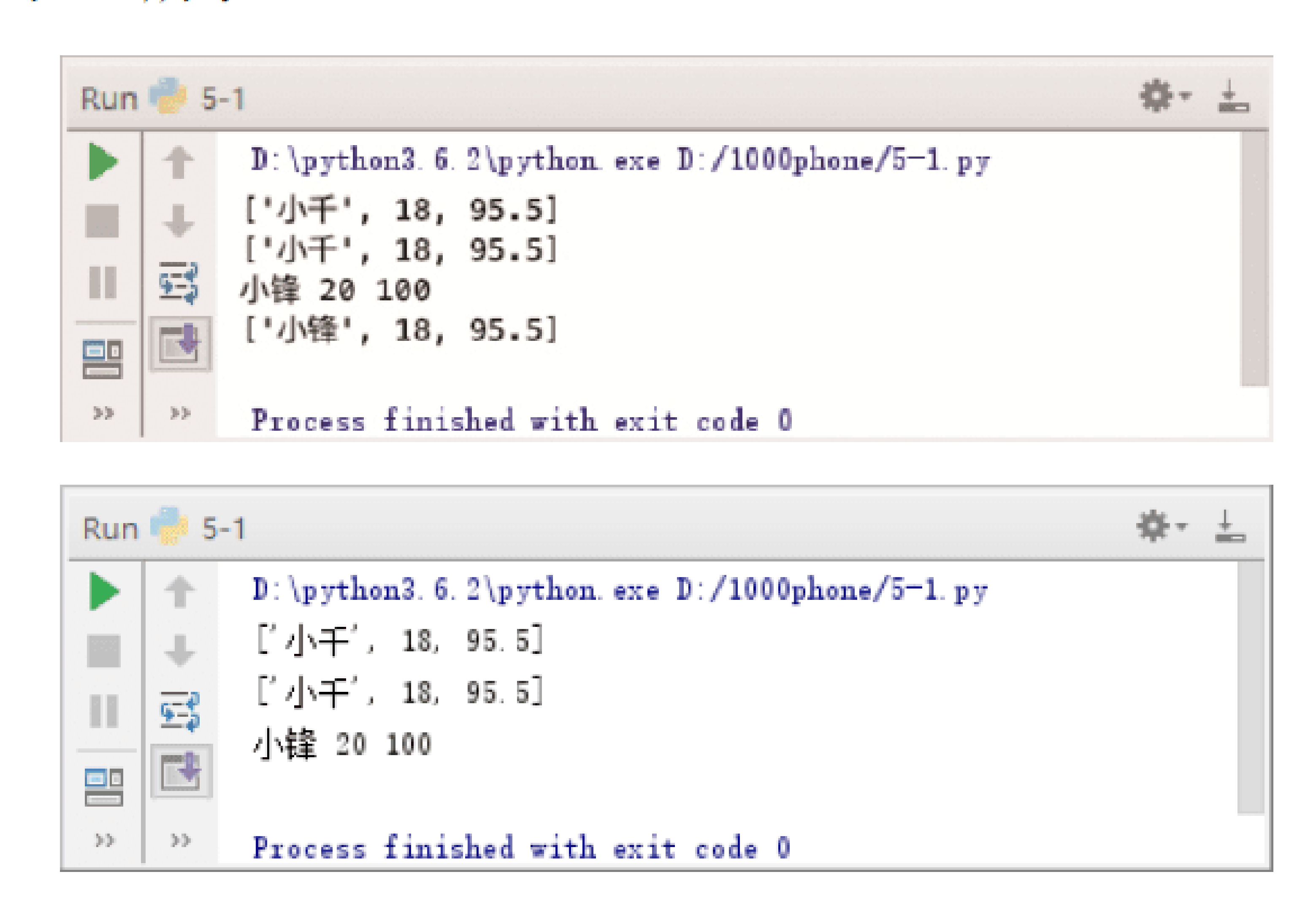


图 5.1 例 5-1 运行结果

在例 5-1 中，通过对比 list1 前后的打印值，可以观察到 list1 列表中的元素值是可以进行更改的（对比第 3 行打印值和第 8 行打印值）。由于 name 是不可变类型（字符串），所以在第 5 行打印 list1 数据时 name 还是更改前的值（涉及数据引用与不可变类型，后续章节将会讲解）。

此外，还可以通过 list()函数创建列表，如例 5-2 所示。

例 5-2 list()函数的用法。

```
list1 = list("qianfeng")
list2 = list(range(1,5))
list3 = list(range(5))
list4 = list(range(1, 5, 2))
print(list1)
print(list2)
print(list3)
print(list4)
```

运行结果如图 5.2 所示。

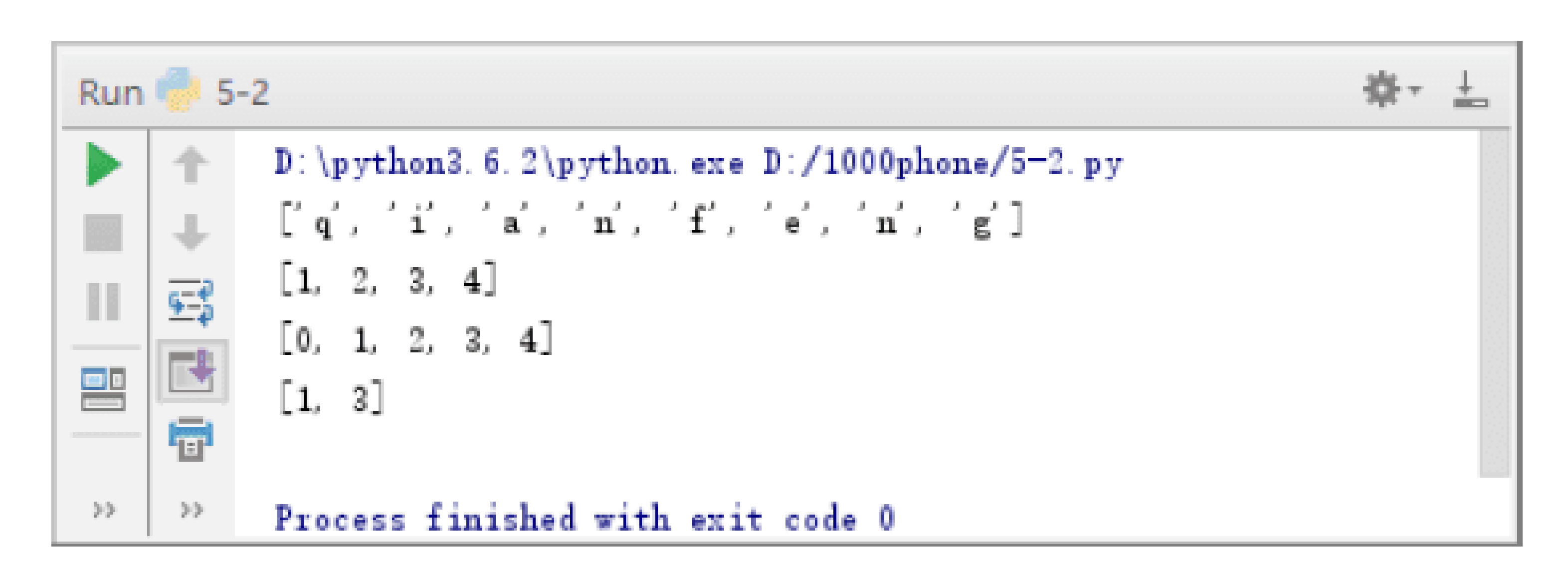


图 5.2　例 5-2 运行结果

在例 5-2 中，第 1 行将字符串中每个字符作为列表中的每个元素。第 2～4 行通过 range()函数生成一系列整数作为列表的元素，range()函数的用法如表 5.1 所示。

表 5.1　range()函数

函　数	说　明
range(start,end)	返回一系列整数从 start 开始，到 end–1 结束，相邻两个整数差 1
range(end)	返回列一系列整数从 0 开始，到 end–1 结束，相邻两个整数差 1
range(start,end,step)	返回一系列整数从 start 开始，相邻两个整数差 step，结束整数不超过 end–1

5.1.2　列表的索引与切片

列表的索引与字符串的索引类似，都分为正向与反向索引，如图 5.3 所示。

list1 = [	1,	2,	3,	4,	5,	6,	7,	8]
index	0	1	2	3	4	5	6	7
	–8	–7	–6	–5	–4	–3	–2	–1

图 5.3　列表索引

在图 5.3 中，列表中每一个元素都对应两个下标，例如索引列表中的元素 5，可以通过以下两种方式指定：

```
list1[4]
list1[-4]
```

列表的切片与字符串的切片也类似，列表的切片可以从列表中取得多个元素并组成一个新列表。接下来演示列表的切片，如例 5-3 所示。

例 5-3　列表的切片。

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8]
print(list1[2:6])
print(list1[2:6:2])
print(list1[:6])
print(list1[2:])
print(list1[-6:-2])
print(list1[-6:-2:2])
print(list1[::-2])
```

运行结果如图 5.4 所示。

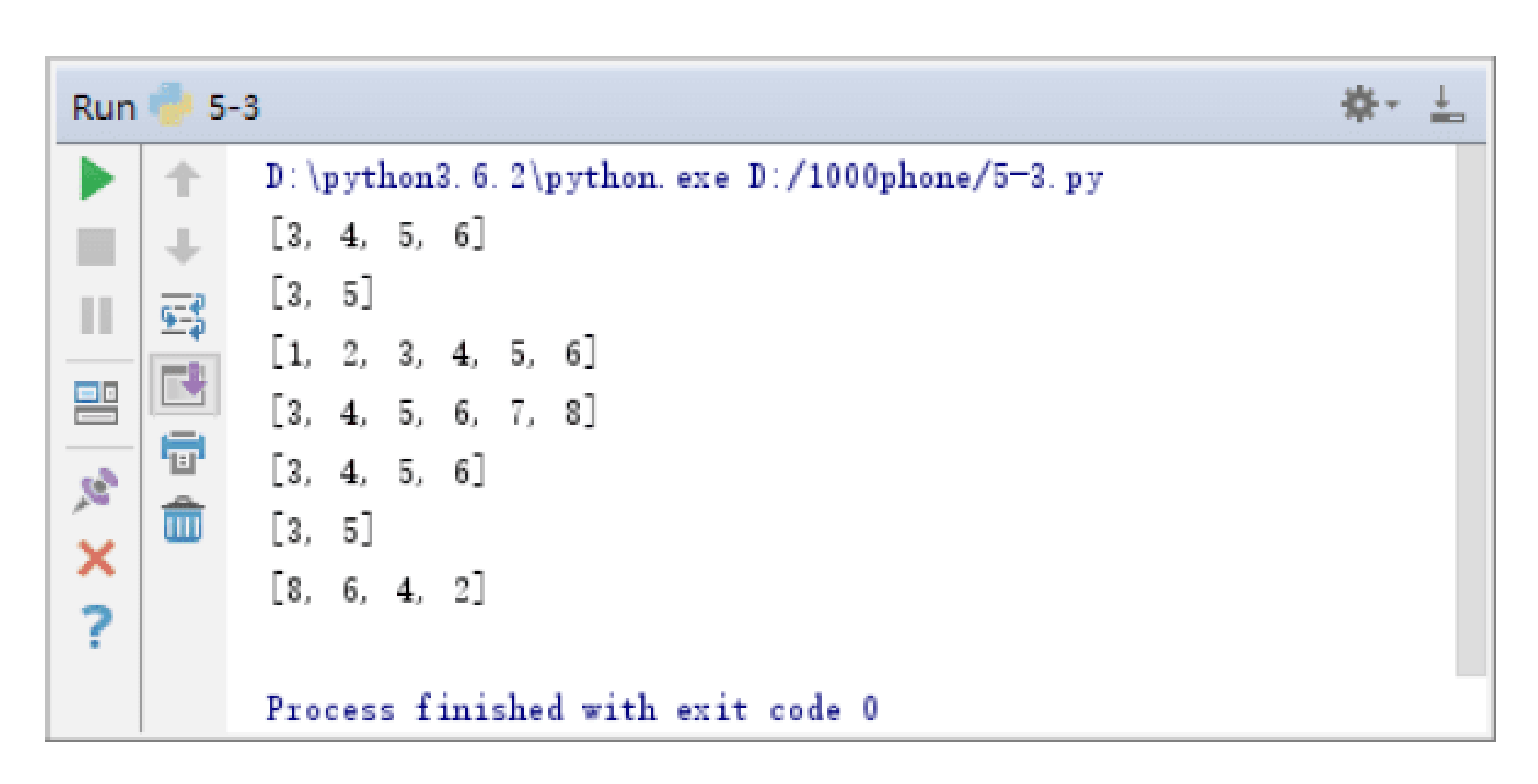


图 5.4 例 5-3 运行结果

在例 5-3 中，值得注意的是，对原列表进行切片操作后返回一个新列表，原列表并没有发生任何变化。

5.1.3 列表的遍历

前两个小节讲解了如何创建列表与索引列表中一个元素，那么如何遍历列表中所有元素？可以通过前面学习的 while 循环或 for 循环实现。

1. 通过 while 循环遍历列表

通过 while 循环遍历列表，需要使用 len()函数，该函数可以获取序列中元素的个数，具体示例如下：

```
print(len('qianfeng'))  # 输出 8
list = [1, 2, 3, 4]
print(len(list))        # 输出 4
```

这样就可以将 len()函数获取列表的个数作为 while 循环的条件，如例 5-4 所示。

例 5-4 通过 while 循环遍历列表。

```
list = ['千锋教育', '扣丁学堂', '好程序员特训营']
length, i = len(list), 0
while i < length:
    print(list[i])
    i += 1
```

运行结果如图 5.5 所示。

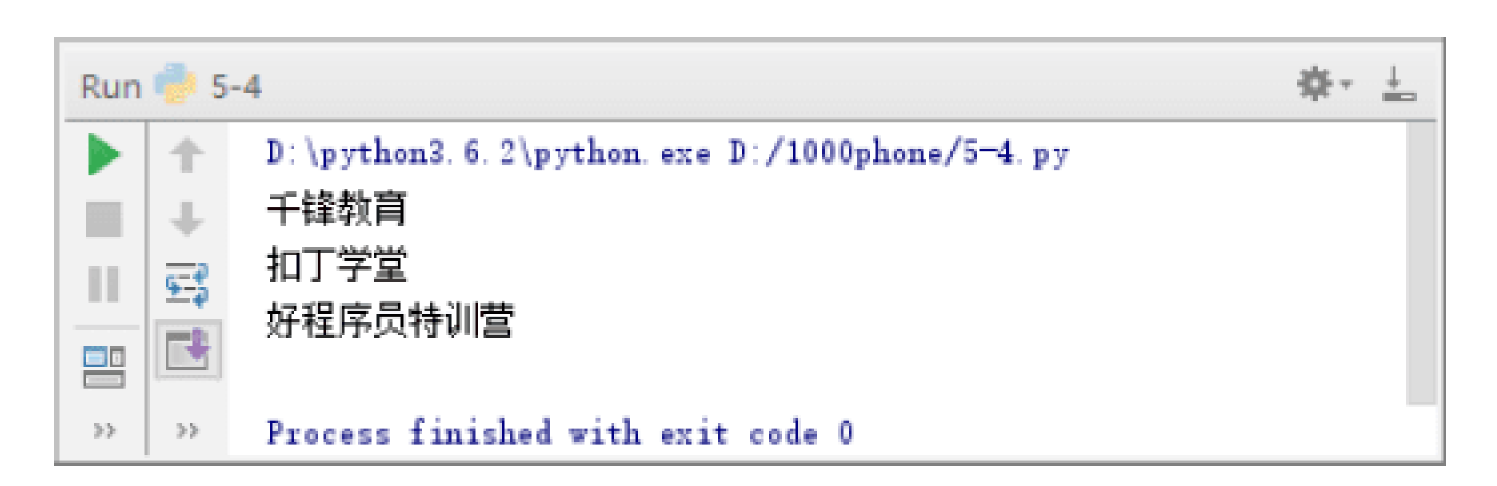


图 5.5 例 5-4 运行结果

在例 5-4 中，while 循环通过控制变量 i 来遍历列表中的元素。

2. 通过 for 循环遍历列表

由于列表是序列的一种，因此通过 for 循环遍历列表非常简单，只需将列表名放在 for 语句中 in 关键词之后即可，如例 5-5 所示。

例 5-5 通过 for 循环遍历列表。

```
list = ['千锋教育', '扣丁学堂', '好程序员特训营']
for value in list:
    print(value)
```

运行结果如图 5.6 所示。

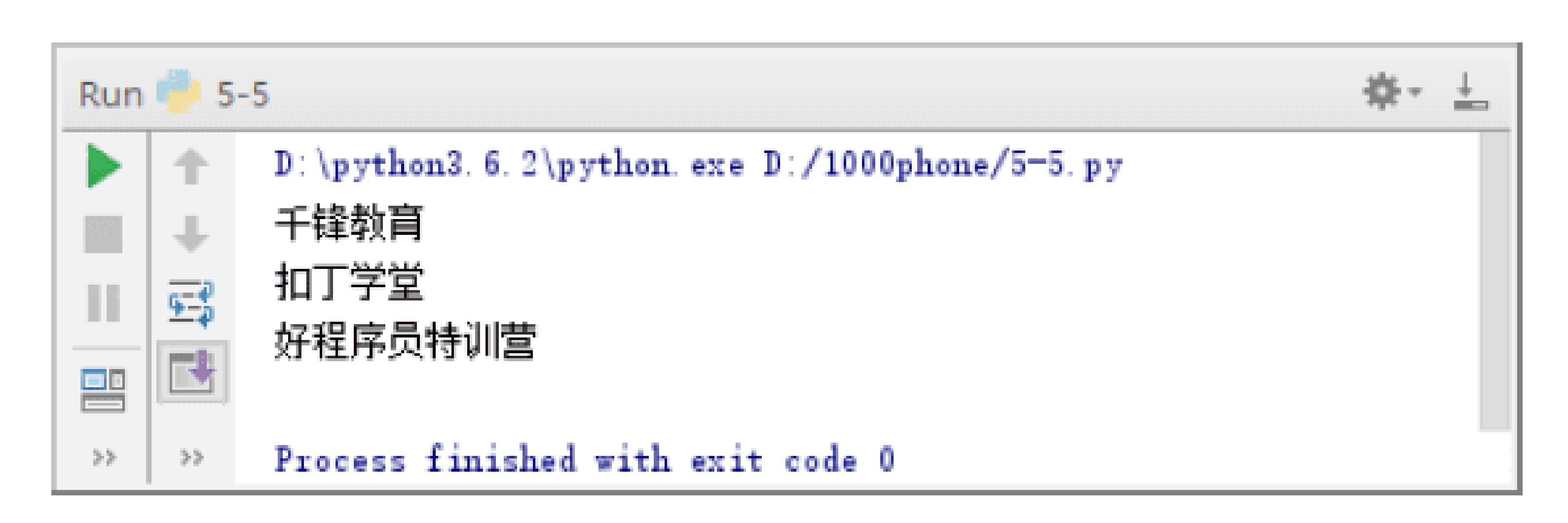


图 5.6 例 5-5 运行结果

在例 5-5 中，for 循环依次将列表中的元素赋值给 value 并通过 print()函数输出。

5.2 列表的运算

列表与字符串类似，也可以进行一些运算，如表 5.2 所示。

表 5.2 列表的运算符

运 算 符	说 明
+	列表连接
*	重复列表元素
[]	索引列表中的元素
[:]	对列表进行切片
in	如果列表中包含给定元素，返回 True
not in	如果列表中包含给定元素，返回 False

接下来演示列表的运算，如例 5-6 所示。

例 5-6 列表的运算。

```
list1, list2 = ['千锋教育', '扣丁学堂'], ['好程序员特训营']
print(list1 + list2)
print(3 * list2)
print("扣丁学堂" in list2)
```

```
print("千锋教育" in list1)
name1, name2 = list1[0:]
name3, name4 = list1
print(name1, name2, name3, name4)
```

运行结果如图 5.7 所示。

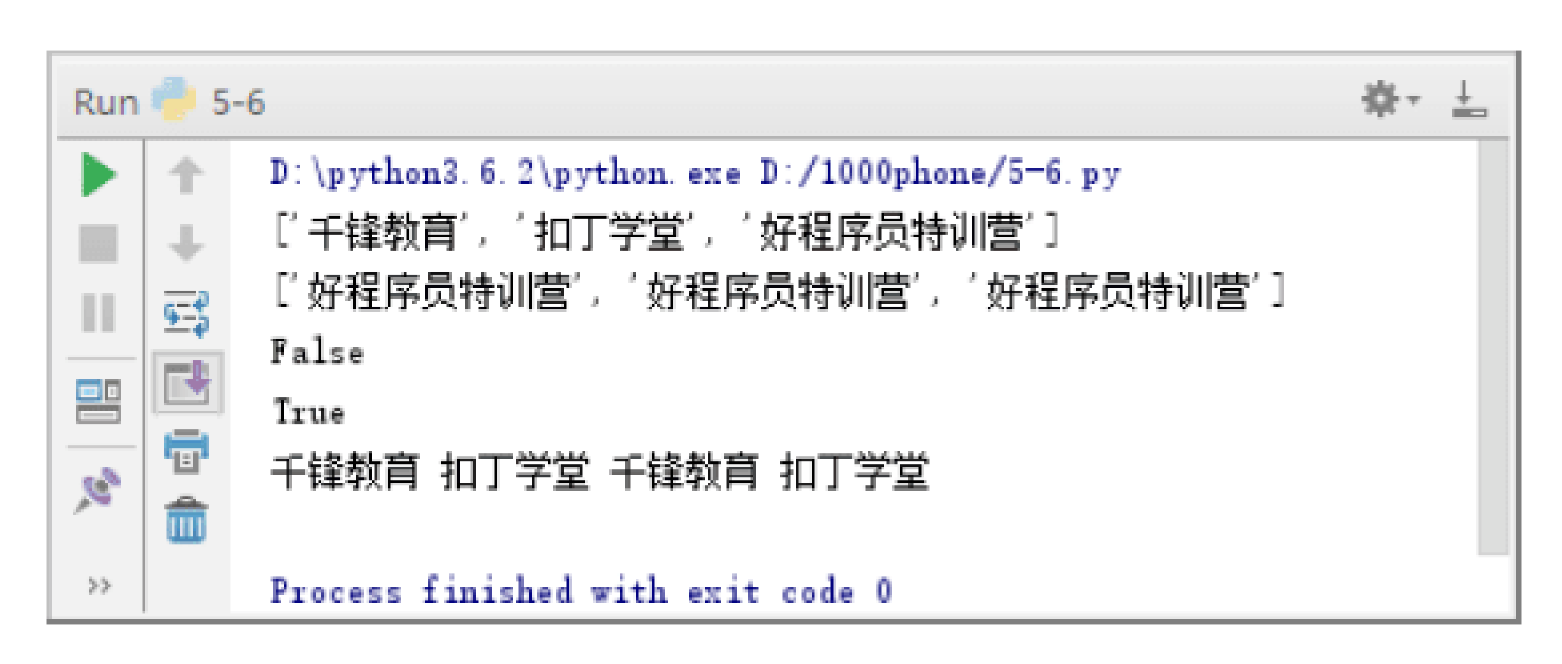


图 5.7 例 5-6 运行结果

在例 5-6 中，程序通过使用列表的运算，可以很方便地操作列表。

5.3 列表的常用操作

列表中存储了不同数据类型的元素，当创建完列表后，就需要对这些元素进行操作，例如添加元素、修改元素、删除元素、元素排序、统计元素个数等。本节讲解列表的常用操作。

5.3.1 修改元素

修改列表中的元素非常简单，只需索引需要修改的元素并对其赋新值即可，如例 5-7 所示。

例 5-7 修改列表中的元素。

```
list1, list2 = ['千锋教育', '扣丁学堂', '好程序员特训营'], [1, 2, 3]
list1[0], list1[1] = 'www.qfedu.com', 'www.codingke.com'
print(list1)
list1[1:] = list2[0:2]
print(list1)
```

运行结果如图 5.8 所示。

```
Run 5-7
D:\python3.6.2\python.exe D:/1000phone/5-7.py
['www.qfedu.com', 'www.codingke.com', '好程序员特训营']
['www.qfedu.com', 1, 2]

Process finished with exit code 0
```

图 5.8 例 5-7 运行结果

在例 5-7 中，第 2 行通过分别对 list[0]、list[1]赋值来改变列表中元素的值，第 4 行通过切片对列表中元素进行赋值。

5.3.2 添加元素

向列表中添加元素的方法有多种，如表 5.3 所示。

表 5.3 添加元素函数

函 数	说 明
append(obj)	在列表末尾添加元素 obj
extend(seq)	在列表末尾一次性添加另一个序列 seq 中的多个元素
insert(index, obj)	将元素 obj 插入列表的 index 位置处

在表 5.3 中，每个函数的作用稍微有点区别。接下来演示其用法，如例 5-8 所示。

例 5-8 向列表中添加元素。

```
1   list1, list2  = [], ['www.qfedu.com', 'www.codingke.com']
2   list1.append('千锋教育')
3   print(list1)
4   list1.extend(list2)
5   print(list1)
6   list1.insert(1, '扣丁学堂')
7   print(list1)
```

运行结果如图 5.9 所示。



图 5.9 例 5-8 运行结果

在例 5-8 中，第 2 行通过 append()函数向空列表 list1 中添加元素'千锋教育'。第 4 行通过 extend()函数向列表 list1 末尾依次添加 list2 中的元素。第 6 行通过 insert()函数向列表 list1 中下标为 1 处添加元素'扣丁学堂'。

5.3.3 删除元素

在列表中删除元素的方法有多种，如表 5.4 所示。

表 5.4 删除元素函数

函 数	说 明
pop(index=–1)	删除列表中 index 处的元素（默认 index=–1），并且返回该元素的值

续表

函　　数	说　　明
remove(obj)	删除列表中第一次出现的 obj 元素
clear()	删除列表中所有元素

接下来演示这 3 个函数的用法，如例 5-9 所示。

例 5-9　在列表中删除元素。

```
1   list  = ['千锋教育', '扣丁学堂', '好程序员特训营', 'qfedu', 'codingke']
2   name = list.pop()
3   print(list, name)
4   name = list.pop(1)
5   print(list, name)
6   list.append('千锋教育')
7   print(list)
8   list.remove('千锋教育')
9   print(list)
10  list.clear()
11  print(list)
```

运行结果如图 5.10 所示。

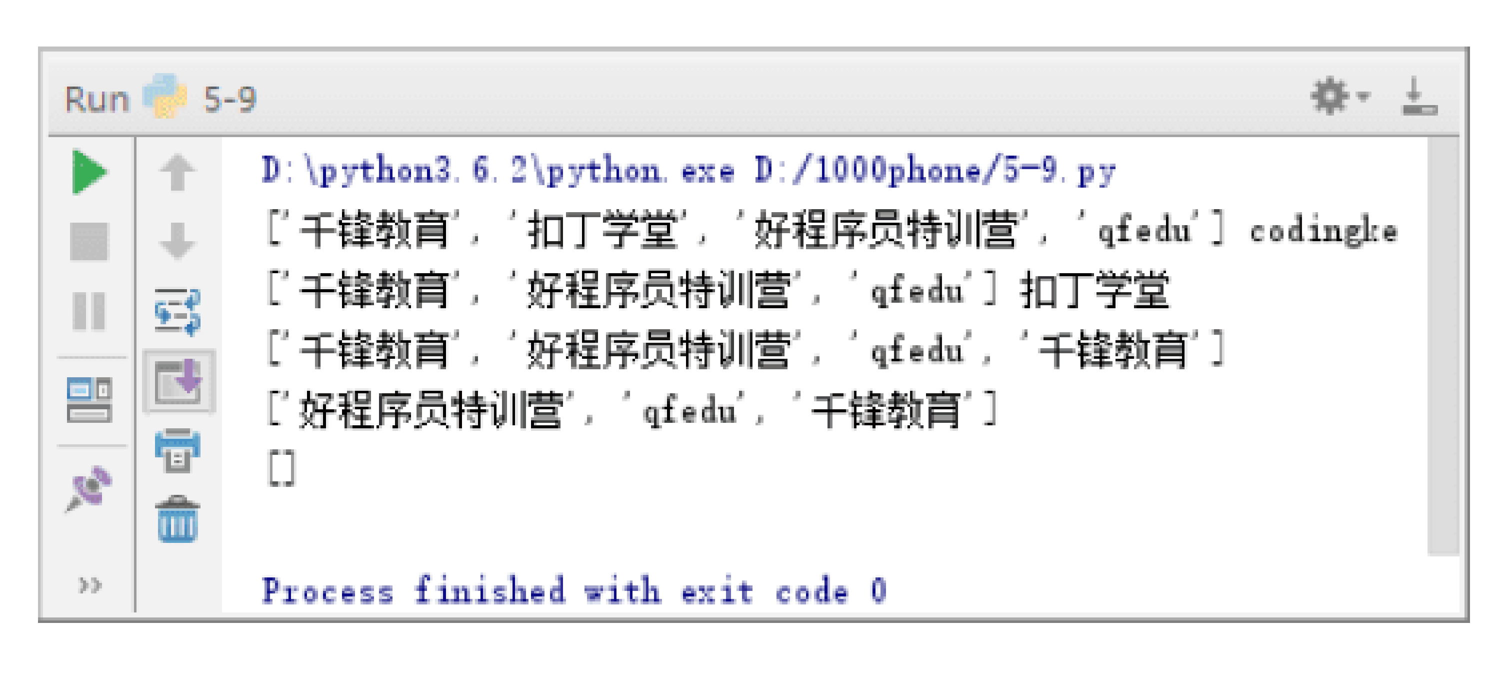


图 5.10　例 5-9 运行结果

在例 5-9 中，第 2 行通过 pop()函数删除列表 list 中最后一个元素并将删除的元素赋值给 name。第 4 行通过 pop()函数删除列表中下标为 1 处的元素并将删除的元素赋值给 name。第 6 行向列表中添加元素'千锋教育'，此时列表中有两个'千锋教育'。第 8 行删除列表中第一次出现的'千锋教育'这个元素。

5.3.4　查找元素位置

index()函数可以从列表中查找出某个元素第一次出现的位置，其语法格式如下：

```
index(obj, start = 0, end = -1)
```

其中，obj 表示需要查找的元素，start 表示查找范围的起始处，end 表示查找范围的结束处（不包括该处）。

接下来演示该函数的用法，如例 5-10 所示。

例 5-10 查找列表中元素的位置。

```
1  list = ['千锋教育', '扣丁学堂', '好程序员特训营', '扣丁学堂']
2  print(list.index('扣丁学堂'))
3  print(list.index('扣丁学堂', 2))
4  print(list.index('扣丁学堂', 1, 3))
```

运行结果如图 5.11 所示。

```
Run 5-10
D:\python3.6.2\python.exe D:/1000phone/5-10.py
1
3
1

Process finished with exit code 0
```

图 5.11 例 5-10 运行结果

在例 5-10 中，第 2 行查找整个列表中'扣丁学堂'第一次出现的位置。第 3 行查找列表下标在[2,−1]范围内'扣丁学堂'第一次出现的位置。第 4 行查找列表下标在[1,3)范围内'扣丁学堂'第一次出现的位置。

5.3.5 元素排序

如果需要对列表中的元素进行排序，则可以使用 sort()函数，如例 5-11 所示。

例 5-11 对列表中的元素进行排序。

```
1  list = [5, 9, 4, 7, 1, 8, 2]
2  list.sort()
3  print(list)
4  list.sort(reverse = True)
5  print(list)
```

运行结果如图 5.12 所示。

```
Run 5-11
D:\python3.6.2\python.exe D:/1000phone/5-11.py
[1, 2, 4, 5, 7, 8, 9]
[9, 8, 7, 5, 4, 2, 1]

Process finished with exit code 0
```

图 5.12 例 5-11 运行结果

在例 5-11 中，第 2 行使用 sort()函数对列表 list 中的元素进行排序，默认按从小到大进行排序。第 4 行设置参数 reverse = True，则列表中的元素按从大到小进行排序。

此外，对列表操作时，reverse()函数可以将列表中的元素反转（也称为逆序），如例 5-12 所示。

例 5-12 对列表中的元素进行反转。

```
1   list = ['千锋教育', '扣丁学堂', '好程序员特训营']
2   list.reverse()
3   print(list)
```

运行结果如图 5.13 所示。

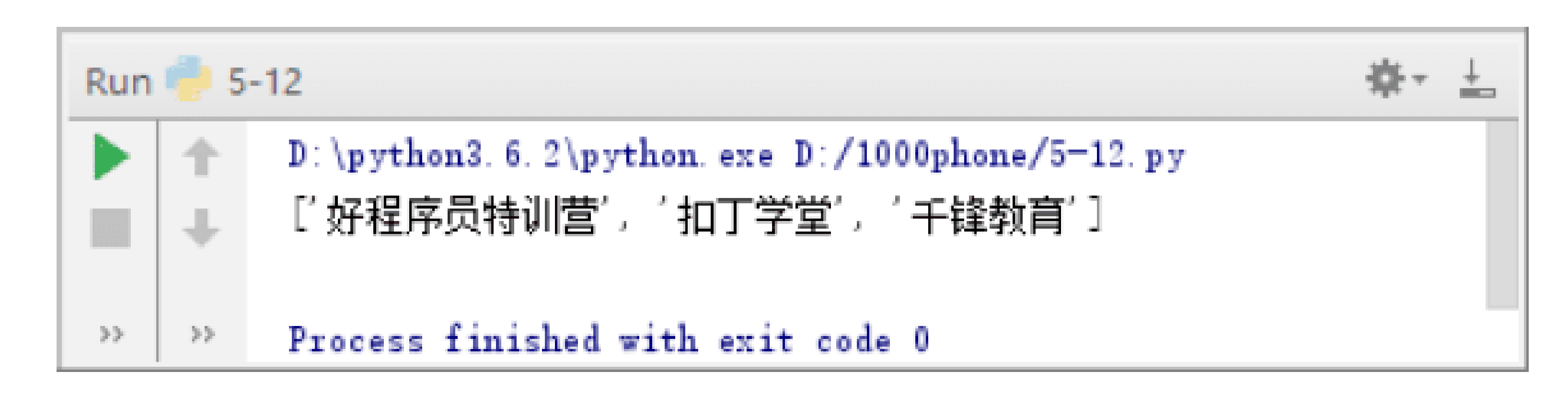


图 5.13 例 5-12 运行结果

在例 5-12 中，第 2 行使用 reverse()函数对列表 list 中的元素进行反转。

5.3.6 统计元素个数

count()函数可以统计列表中某个元素的个数，如例 5-13 所示。

例 5-13 统计列表中某个元素的个数。

```
1   list = ['千锋教育', '扣丁学堂', '好程序员特训营', '扣丁学堂']
2   print(list.count('扣丁学堂'))
```

运行结果如图 5.14 所示。

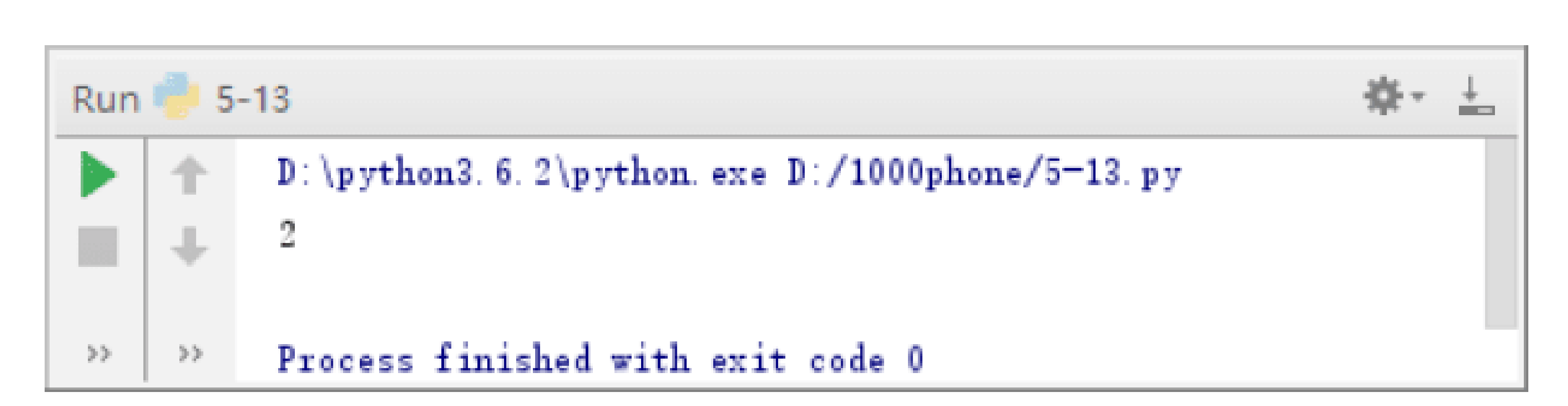


图 5.14 例 5-13 运行结果

在例 5-13 中，第 2 行使用 count()函数统计列表 list 中元素'扣丁学堂'的个数。

5.4 列表推导

根据前面学习的知识，已有一个包含 10 个整数的列表 list，创建一个新列表 newList，该列表中每个元素为 list 列表中每个元素的平方，如例 5-14 所示。

例 5-14 newList 列表中每个元素为 list 列表中每个元素的平方。

```
list = range(1, 11)
```

```
newList = []
for num in list:
    newList.append(num ** 2)
print(newList)
```

运行结果如图 5.15 所示。

```
Run 5-14
D:\python3.6.2\python.exe D:/1000phone/5-14.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Process finished with exit code 0
```

图 5.15 例 5-14 运行结果

在例 5-14 中，通过 for 循环遍历 list 中的每一个元素并计算出平方值，然后将平方值添加到列表 newList 中。

在 Python 中可以使用更简单的方法实现上述功能，如例 5-15 所示。

例 5-15 列表推导。

```
list = range(1, 11)
newList = [num ** 2 for num in list]
print(newList)
```

运行结果如图 5.16 所示。

```
Run 5-15
D:\python3.6.2\python.exe D:/1000phone/5-15.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Process finished with exit code 0
```

图 5.16 例 5-15 运行结果

在例 5-15 中，仅使用一行语句就完成例 5-14 中 3 行语句的功能，其中用到的知识就是列表推导，其语法格式如下：

```
[表达式 1 for k in L if 表达式 2]
```

该语句与下面的语句等价，具体如下所示：

```
List = []
for k in L:
    if 表达式 2:
        List.append(表达式 1)
```

其中，List 的元素由每一个“表达式 1”组成。if 语句用于过滤，可以省略。

接下来演示列表推导中含有 if 语句，如例 5-16 所示。

例 5-16 列表推导中含有 if 语句。

```
list = range(1, 11)
newList = [num ** 2 for num in list if num > 5]
print(newList)
```

运行结果如图 5.17 所示。

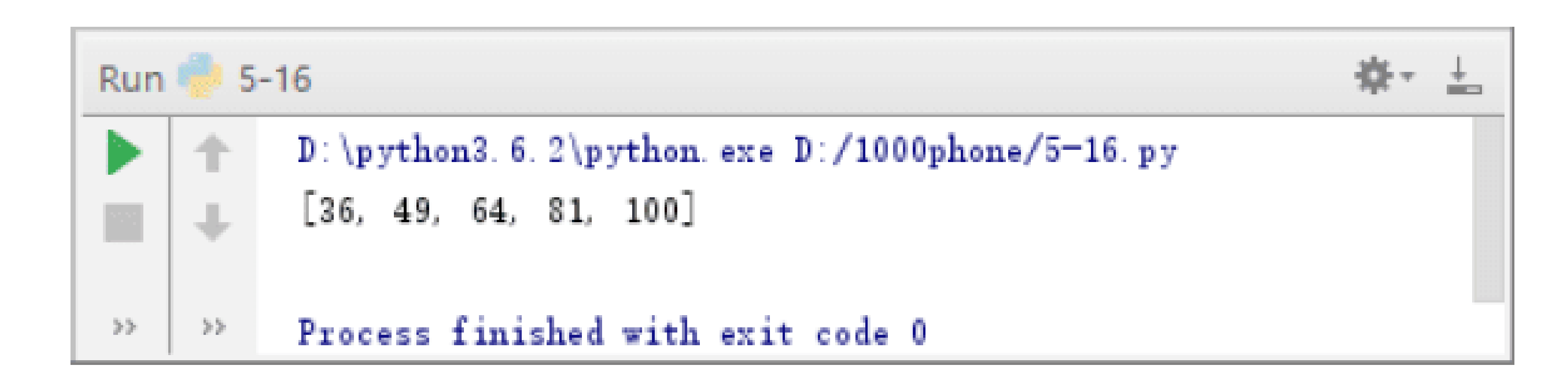


图 5.17　例 5-16 运行结果

在例 5-16 中，通过 if 条件语句过滤 list 列表中大于 5 的元素值，然后对该值进行平方并加入到 newList 列表中。

5.5　元　　组

元组与列表类似，也是一种序列，不同之处在于元组中元素不能被改变，并且使用小括号中的一系列元素。

5.5.1　元组的创建

创建元组的语法非常简单，只需用逗号将元素隔开，具体示例如下：

```
tuple1 = 1, 2, 3, 4
tuple2 = 'xiaoqian', 18, 100
```

通常是通过小括号将元素括起来，具体示例如下：

```
tuple3 = (1, 2, 3, 4)
tuple4 = ('xiaoqian', 18, 100)
```

此外，还可以创建一个空元组，具体示例如下：

```
tuple5 = ()
```

接下来创建只包含一个元素的元组，创建方式有些特别，具体示例如下：

```
tuple6 = (1,)
```

注意此处逗号必须添加，如果省略，则相当于在一个小括号内输入了一个值。此处添加逗号后，就通知解释器，这是一个元组，具体示例如下：

```
tuple6 = (1,)
tuple7 = (1)
```

如果通过 print()函数将 tuple6 与 tuple7 分别进行输出，则得到以下结果：

```
(1,)
1
```

通过输出结果可得出，tuple6 为元组，tuple7 为一个整数。

5.5.2 元组的索引

元组可以使用下标索引来访问元组中的一个元素，也可以使用切片访问多个元素，如例 5-17 所示。

例 5-17 元组的索引。

```
1  tuple = ('千锋教育', '扣丁学堂', '好程序员特训营')
2  print(tuple[0])
3  tuple1 = tuple[0:-1]
4  print(tuple1)
5  tuple2 = tuple[1:]
6  print(tuple2)
```

运行结果如图 5.18 所示。

图 5.18 例 5-17 运行结果

在例 5-17 中，第 2 行通过下标索引访问元组中的元素。第 3 行与第 5 行对元组进行切片，元组的切片还是元组，就像列表的切片还是列表一样。

注意不能通过下标索引修改元组中的元素，具体示例如下：

```
tuple[0] = 'www.qfedu.com'  # 错误
```

上述语句运行时会报错，因为元组中的元素不能被修改。

初学者学习元组时，可能会疑惑既然有列表，为什么还需要元组，原因如下：

- 元组的速度比列表快。如果定义了一系列常量值，而所做的操作仅仅是对它进行遍历，那么一般使用元组而不是列表。
- 元组对需要修改的数据进行写保护，这样将使得代码更加安全。
- 一些元组可用作字典键。

5.5.3 元组的遍历

元组的遍历与列表的遍历类似，都可以通过 for 循环实现，如例 5-18 所示。

例 5-18 元组的遍历。

```
tuple = ('千锋教育', '扣丁学堂', '好程序员特训营')
for name in tuple:
    print(name)
```

运行结果如图 5.19 所示。

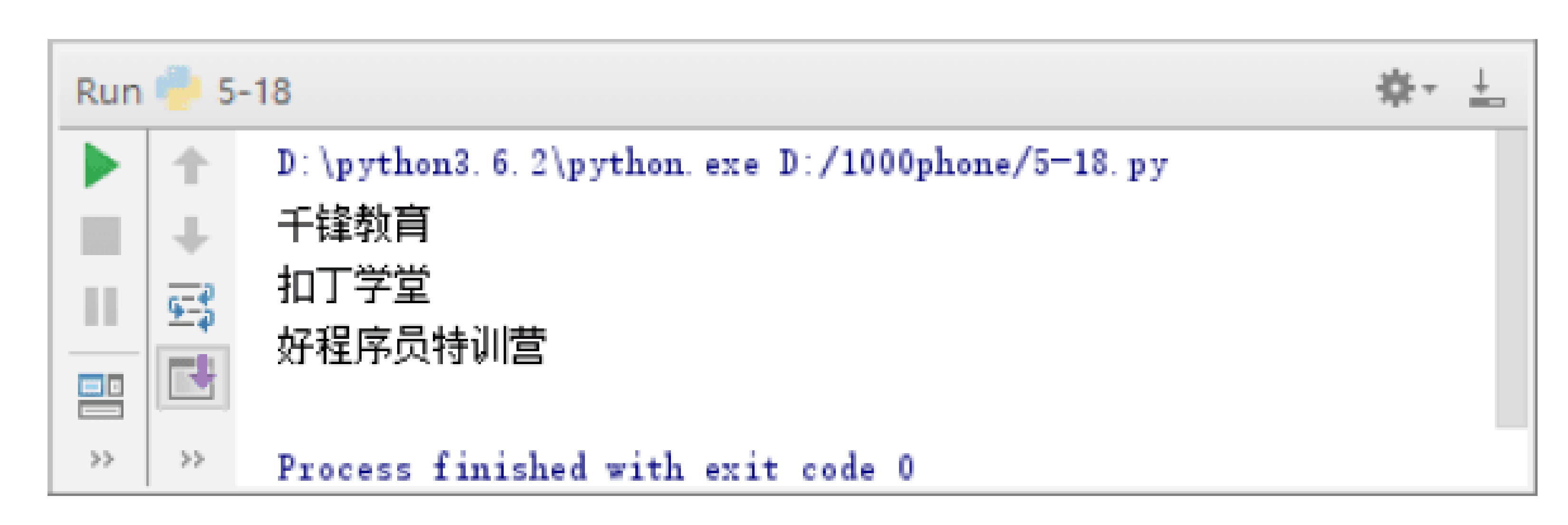


图 5.19 例 5-18 运行结果

在例 5-18 中，for 循环依次将列表中的元素赋值给 name 并通过 print()函数输出。

5.5.4 元组的运算

元组的运算与列表的运算类似，如例 5-19 所示。

例 5-19 元组的运算。

```
tuple1 = ('千锋教育', '扣丁学堂', '好程序员特训营')
tuple2 = ('qfedu', 'codingke')
print(tuple1 + tuple2)
print(tuple2 * 3)
print('千锋教育' in tuple1)
print('扣丁学堂' not in tuple2)
```

运行结果如图 5.20 所示。

```
Run 5-19
D:\python3.6.2\python.exe D:/1000phone/5-19.py
('千锋教育', '扣丁学堂', '好程序员特训营', 'qfedu', 'codingke')
('qfedu', 'codingke', 'qfedu', 'codingke', 'qfedu', 'codingke')
True
True

Process finished with exit code 0
```

图 5.20 例 5-20 运行结果

5.5.5 元组与列表转换

list()函数可以将元组转换为列表，而 tuple()函数可以将列表转换为元组，如例 5-20

所示。

例 5-20　元组与列表的转换。

```
1    tuple1 = ('千锋教育', '扣丁学堂', '好程序员特训营')
2    list1 = list(tuple1)
3    print(list1)
4    tuple2 = tuple(list1)
5    print(tuple2)
```

运行结果如图 5.21 所示。



图 5.21　例 5-20 运行结果

在例 5-20 中，第 2 行通过 list()函数将元组 tuple1 转换为列表 list1，第 4 行通过 tuple()函数将列表 list1 转换为元组 tuple2。

5.6 小　案　例

5.6.1 案例一

在某比赛中，共有 5 位评委给选手打分。计算选手得分时，去掉最高分与最低分，然后求其平均值，该值就是选手的得分，具体实现如例 5-21 所示。

例 5-21　求选手平均分。

```
score = []
for i in range(1,6):
    num = float(input('%d 号评委打分：'%i))
    score.append(num)
min = min(score)                    # 获取最低分
max = max(score)                    # 获取最高分
score.remove(min)                   # 去除最低分
score.remove(max)                   # 去除最高分
ave = sum(score) / len(score)       # 求平均值
print('选手最终得分为%.2f'%ave)
```

运行结果如图 5.22 所示。

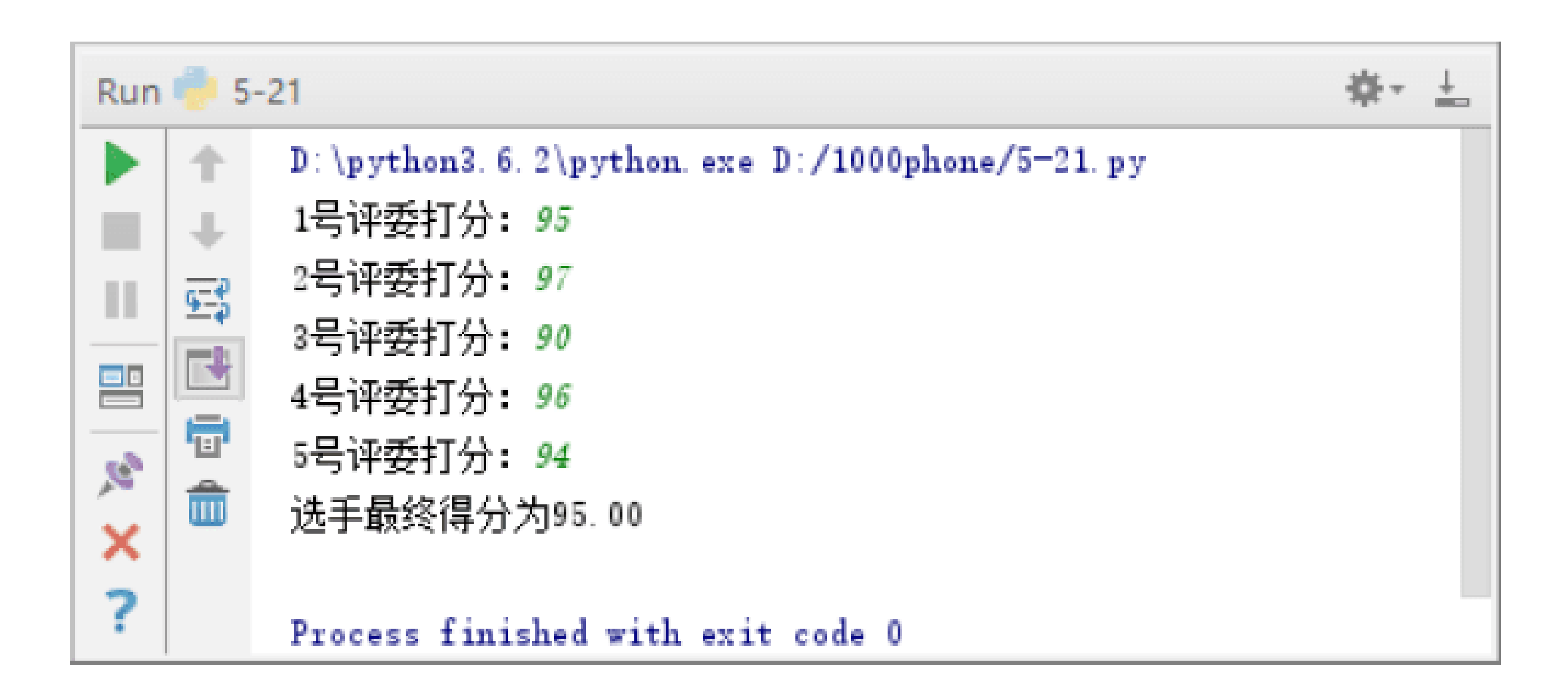


图 5.22　例 5-21 运行结果

在例 5-21 中，首先定义一个空列表，然后通过 for 循环依次往列表 score 中添加元素，直到 5 位评委得分都输入结束循环，接着从列表 score 中移除最大值与最小值，最后通过 sum()函数求和并除以元素个数得到平均分。

5.6.2　案例二

在 Python 中，矩阵可以用列表来表示，具体示例如下：

```
matrix = [
    [1, 3, 5],
    [2, 6, 8],
    [7, 9, 4]
]
```

示例中代表的矩阵如下所示：

$$matrix = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 6 & 8 \\ 7 & 9 & 4 \end{pmatrix}$$

现要求通过代码将该矩阵进行转置，即变为如下形式：

$$matrix^T = \begin{pmatrix} 1 & 2 & 7 \\ 3 & 6 & 9 \\ 5 & 8 & 4 \end{pmatrix}$$

具体实现过程，如例 5-22 所示。

例 5-22　使用列表实现矩阵。

```
matrix = [
    [1, 3, 5],
    [2, 6, 8],
    [7, 9, 4]
]
print(matrix)
# 方法一
newMatrix = []
```

```
for i in range(len(matrix[0])):
    newRow = []
    for oldRow in matrix:
        newRow.append(oldRow[i])
    newMatrix.append(newRow)
print(newMatrix)
# 方法二
newMatrix = []
for i in range(len(matrix[0])):
    newMatrix.append([row[i] for row in matrix])
print(newMatrix)
# 方法三
newMatrix = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(newMatrix)
```

运行结果如图 5.23 所示。

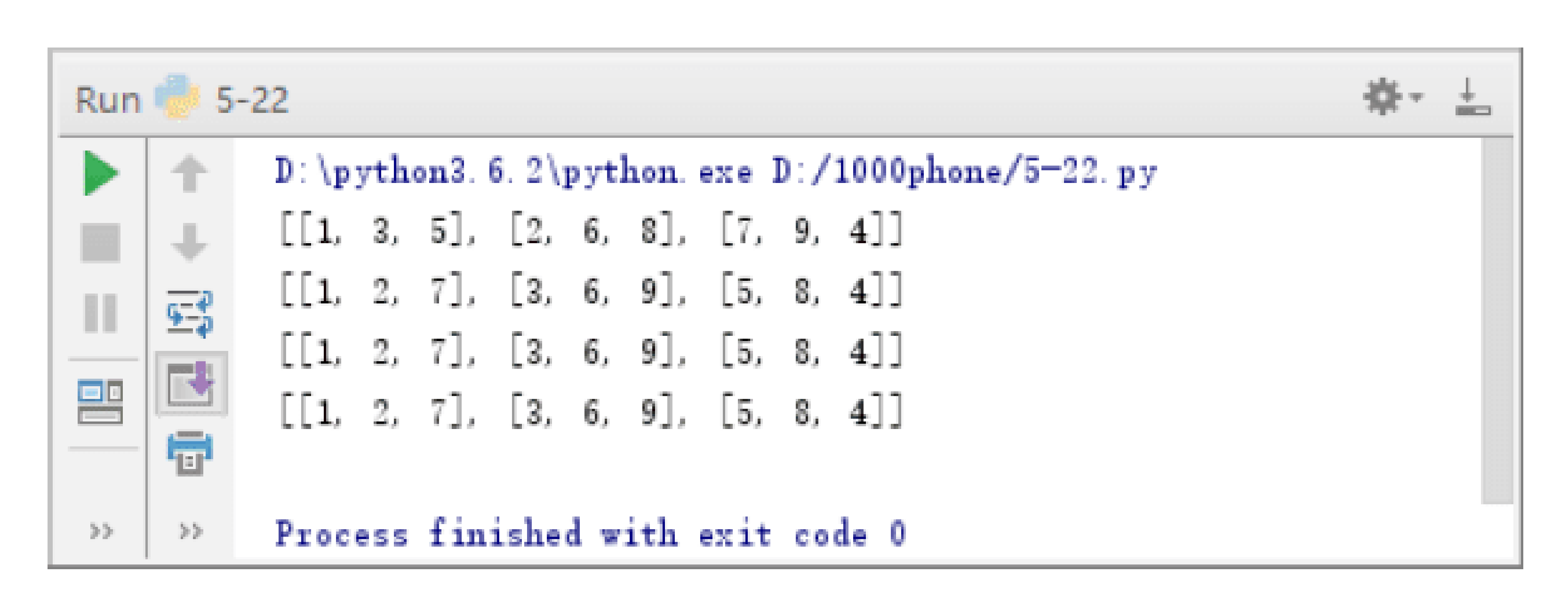


图 5.23 例 5-22 运行结果

在例 5-22 中，矩阵的表示方法可以看成是列表嵌套列表。第一种方法通过两层 for 循环完成矩阵的转置，每次从列表 matrix 的所有子列表中取一个元素作为另一个列表 newMatrix 的子列表。第二种方法与第三种方法使用了列表推导，本质还是 for 循环。

5.7 本章小结

本章主要介绍了 Python 中的列表与元组，两者都是序列。列表使用中括号表示，其中的元素可以被修改，而元组使用小括号表示，其中的元素不能被修改。在实际开发中，应根据这两种序列的特点选择合适的类型。

5.8 习题

1. 填空题

（1）列表使用________括号表示。

（2）元组使用________括号表示。
（3）________函数可以删除列表中最后一个元素。
（4）________函数可以对列表中的元素进行排序。
（5）________函数可以将列表转换为元组。

2．选择题

（1）下列属于列表的是（　　）。
A．1, 2, 3, 4　　B．[1, 2, 3, 4]　　C．{1, 2, 3, 4}　　D．(1, 2, 3, 4)
（2）若 list = [2, 3, 1, 4]，在经过 list.reverse()操作后，list 为（　　）。
A．(4, 1, 3, 2)　　B．(3, 2, 4, 1)　　C．[4, 1, 3, 2]　　D．[3, 2, 4, 1]
（3）下列不属于元组的是（　　）。
A．'a', 'b', 'c'　　B．1, 2, 3　　C．['a', 'b', 'c']　　D．(1, 2, 3)
（4）若 tuple = (2, 3, 1, 4)，则 list(tuple)返回（　　）。
A．[2, 3, 1, 4]　　B．(2, 3, 1, 4)　　C．2, 3, 1, 4　　D．None
（5）若 a = (2)，则 print(a)输出（　　）。
A．(2, 0)　　B．(2,)　　C．None　　D．2



3．思考题

（1）简述列表与元组的区别。
（2）若 a = [1]，则 a.append(['a', 'b'])与 a.extend(['a', 'b'])实现的效果一样吗？

4．编程题

水仙花数是指一个 n 位数（n≥3），它的每位上的数字的 n 次幂之和等于它本身。例如：$1^3+5^3+3^3=153$。求 100～999 之间所有的水仙花数。

第 6 章 chapter 6

字典与集合

本章学习目标

- 理解字典的概念。
- 掌握字典的创建。
- 掌握字典的常用操作。
- 了解集合的概念。
- 了解集合的常用操作。

列表与元组都是通过下标索引元素，由于下标不能代表具体的含义，为此 Python 提供了另一种数据类型——字典，这为编程带来了极大的便利。此外，Python 还提供了一种数据类型——集合，其最大特点是元素不能重复出现，因此通常用来处理元素的去重操作。

6.1 字典的概念

在现实生活中，字典可以查询某个词的语义，即词与语义建立了某种关系，通过词的索引便可以找到对应的语义，如图 6.1 所示。

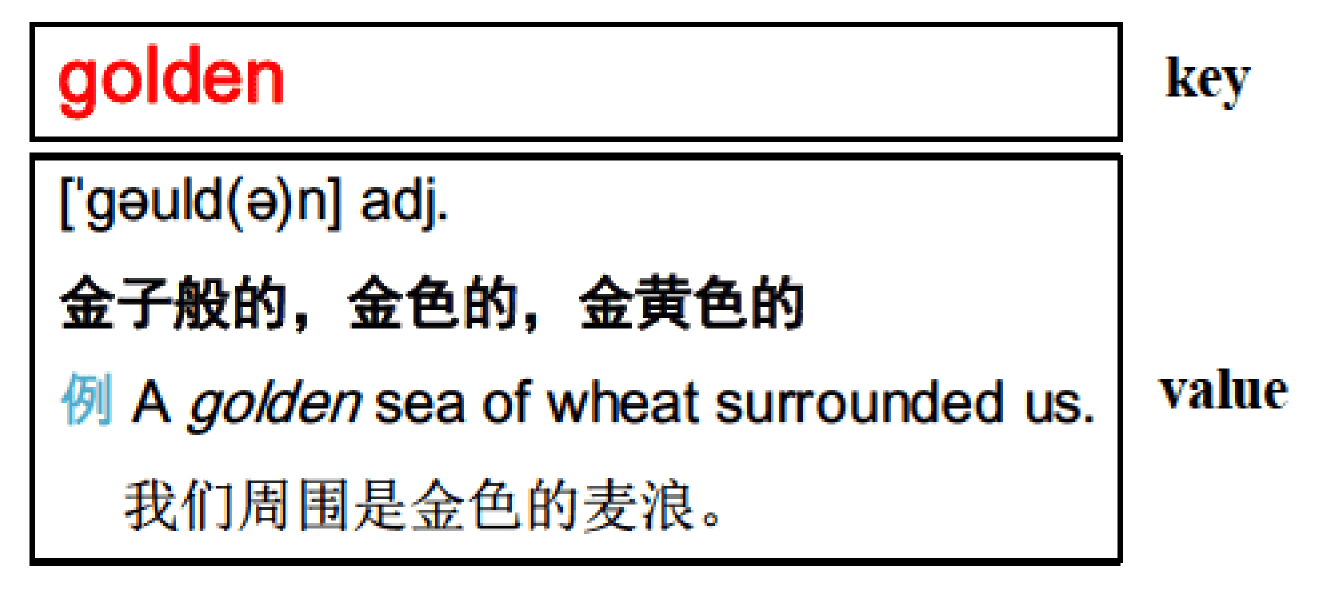


图 6.1 字典

在 Python 中，字典也如现实生活中的字典一样，使用词-语义进行数据的构建，其中词对应键（key），词义对应值（value），即键与值构成某种关系，通常将两者称为键值对，这样通过键可以快速找到对应的值。

字典是由元素构成的，其中每个元素都是一个键值对，具体示例如下：

```
student = {'name': '小千', 'id': 20190101, 'score': 98.5}
```

示例中，字典由 3 个元素构成，元素之间用逗号隔开，整体用大括号括起来。每个元素是一个键值对，键与值之间用冒号隔开，如'name':'xiaoqian'，'name'是键，'xiaoqian'是值。

因为字典是通过键来索引值的，所以键必须是唯一的，而值并不唯一，具体示例如下：

```
student = {'name': '小千', 'name': '小锋', 'score1': 98.5, 'score2': 98.5}
```

示例中，字典中有两个元素的键为'name'，有两个元素的值为 98.5。若通过 print(student)输出字典，则得到以下输出结果：

```
{'name': '小锋', 'score1': 98.5, 'score2': 98.5}
```

从上述结果可看出，如果字典中存在相同键的元素，那么只会保留后面的元素。

另外，键不能是可变数据类型，如列表，而值可以是任意数据类型，具体示例如下：

```
student = {['name', 'alias']: '小千'}  # 错误
```

上述语句在程序运行时会引发错误。

通过上面的学习，可以总结出字典具有以下特征：

- 字典中的元素是以键值对的形式出现的。
- 键不能重复，而值可以重复。
- 键是不可变数据类型，而值可以是任意数据类型。

6.2 字典的创建

了解了字典的概念后，接下来创建一个字典，具体示例如下：

```
dict1 = {}
```

上述语句创建了一个空字典，也可以在创建字典时指定其中的元素，具体示例如下：

```
dict2 = {'name': '小千', 'id': 20190101, 'score': 98.5}
```

字典中值可以取任何数据类型，但键必须是不可修改的，如字符串、元组，具体示例如下：

```
dict3 = {20190101: ['小千', 100], (1101, '大一'):['小锋', 99]}
```

此外，还可以使用 dict()创建字典，如例 6-1 所示。

例 6-1 使用 dict()创建字典。

```
items = [('name', '小千'), ('score', 98)] # 列表
d = dict(items)
print(d)
```

运行结果如图 6.2 所示。

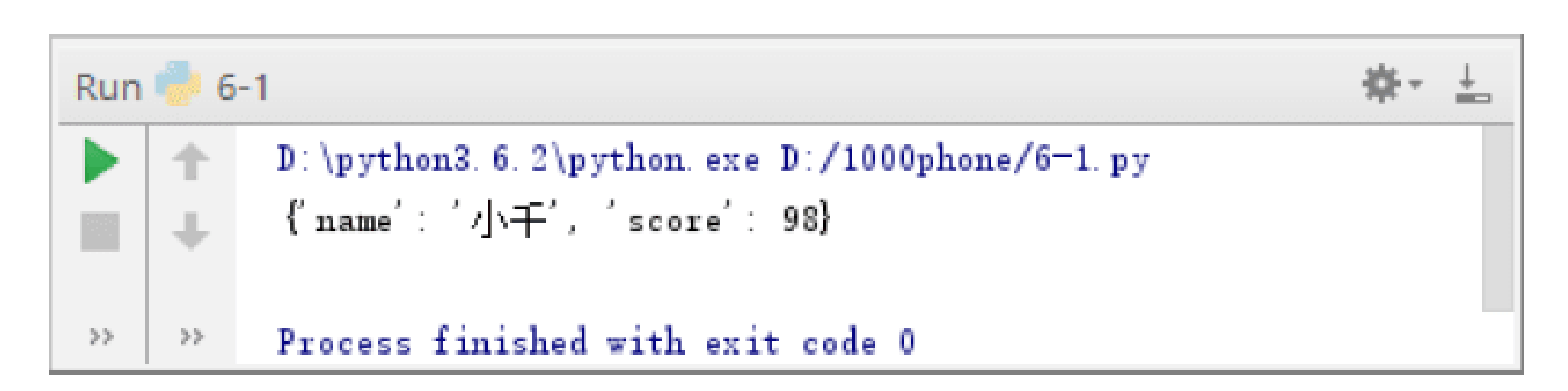


图 6.2　例 6-1 运行结果

在例 6-1 中，第 1 行定义一个列表，列表中的每个元素为元组。第 2 行通过 dict()将列表转换为字典并赋值给 d。

此外，dict()还可以通过设置关键字参数创建字典，如例 6-2 所示。

例 6-2　dict()通过设置关键字参数创建字典。

```
1    d = dict(name = '小千', score = 98)
2    print(d)
```

运行结果如图 6.3 所示。

```
Run 6-2
D:\python3.6.2\python.exe D:/1000phone/6-2.py
{'name': '小千', 'score': 98}

Process finished with exit code 0
```

图 6.3　例 6-2 运行结果

在例 6-2 中，第 1 行通过设置 dict()中参数来指定创建字典的键值对。

6.3　字典的常用操作

在实际开发中，字典使得数据表示更加完整，因此它是应用最广的一种数据类型。想要熟练运用字典，就必须熟悉字典中常用的操作。

6.3.1　计算元素个数

字典中元素的个数可以通过 len()函数来获取，如例 6-3 所示。

例 6-3　通过 len()函数获取字典中元素的个数。

```
dict = {'qfedu':'千锋教育', 'codingke':'扣丁学堂'}
print(len(dict))
```

运行结果如图 6.4 所示。

```
Run 6-3
D:\python3.6.2\python.exe D:/1000phone/6-3.py
2

Process finished with exit code 0
```

图 6.4 例 6-3 运行结果

在例 6-3 中，第 2 行通过 len()函数计算元素个数并通过 print()函数输出。

6.3.2 访问元素值

列表与元组是通过下标索引访问元素值，字典则是通过元素的键来访问值，如例 6-4 所示。

例 6-4 访问字典的元素值。

```
1   dict = {'qfedu':'千锋教育', 'codingke':'扣丁学堂'}
2   print(dict['qfedu'])
3   print(dict['codingke'])
```

运行结果如图 6.5 所示。

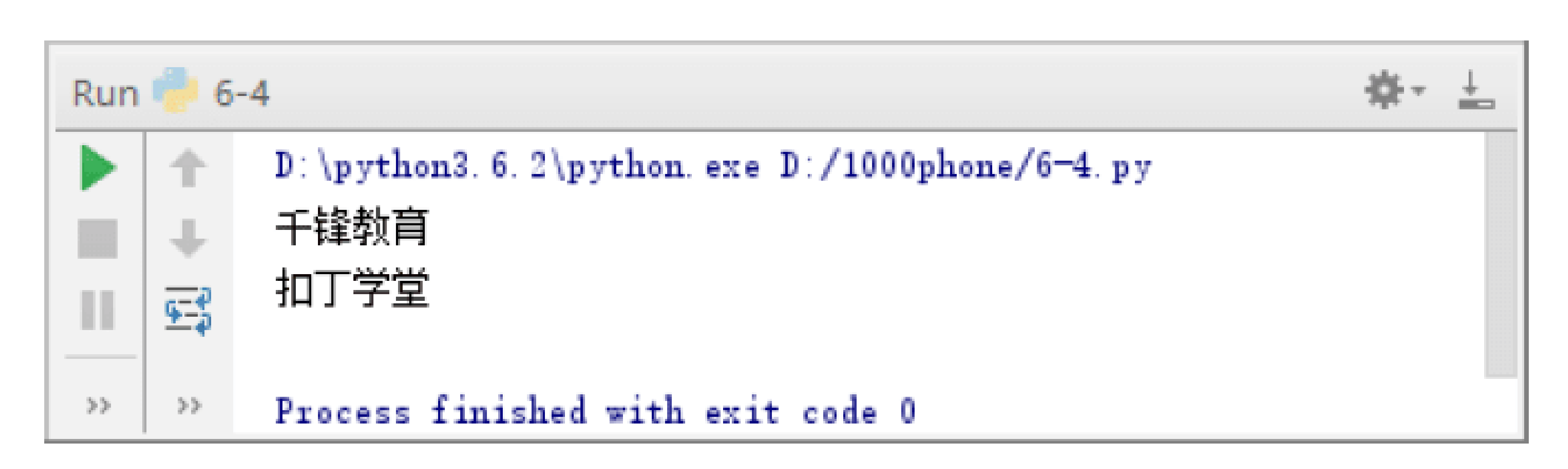


图 6.5 例 6-4 运行结果

在例 6-4 中，第 2 行与第 3 行通过键访问所对应的值并通过 print()函数输出。如果访问不存在的键，则运行时程序会报错。

有时不确定字典中是否存在某个键而又想访问该键对应的值，则可以通过 get()函数实现，如例 6-5 所示。

例 6-5 get()函数的用法。

```
1   dict = {'qfedu':'千锋教育', 'codingke':'扣丁学堂'}
2   name1 = dict.get('goodProgrammer')  # 不存在该键时,返回 None,而不是报错
3   print(name1)
4   name2 = dict.get('qfedu')           # 存在该键时,返回对应的值
5   print(name2)
6   name3 = dict.get('1000phone', '千锋') # 不存在该键时,返回指定值,即第二个参数
7   print(name3)
```

运行结果如图 6.6 所示。

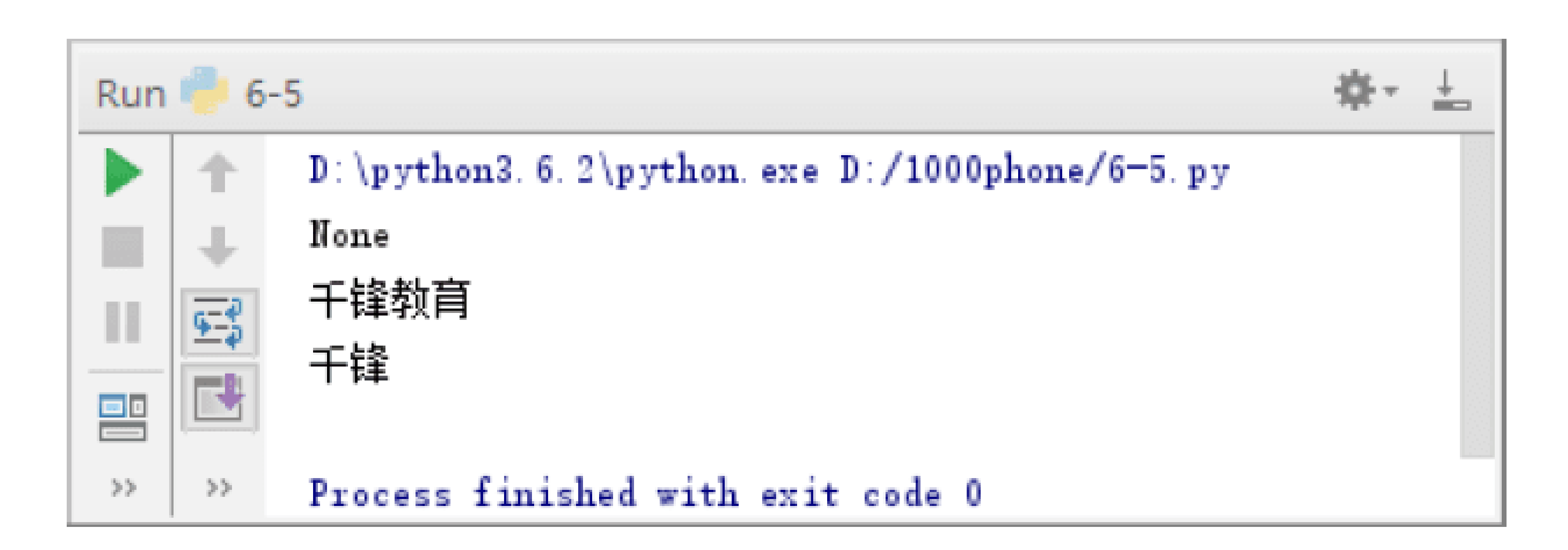


图 6.6　例 6-5 运行结果

在例 6-5 中，第 2 行通过 get()函数获取'goodProgrammer'对应的值，字典中不存在这个键，此时返回 None，而不是报错。第 4 行通过 get()函数获取'qfedu'对应的值，字典中存在这个键，此时返回'千锋教育'。第 6 行通过 get()函数获取'1000phone'对应的值，字典中不存在这个键，此时返回指定值'千锋'。

6.3.3　修改元素值

字典中除了通过键访问值外，还可以通过键修改值，如例 6-6 所示。

例 6-6　修改字典中元素的值。

```
1    std = {'name':'小千', 'score':100}
2    print(std)
3    std['name'] = '小锋'
4    std['score'] = 99
5    print(std)
```

运行结果如图 6.7 所示。

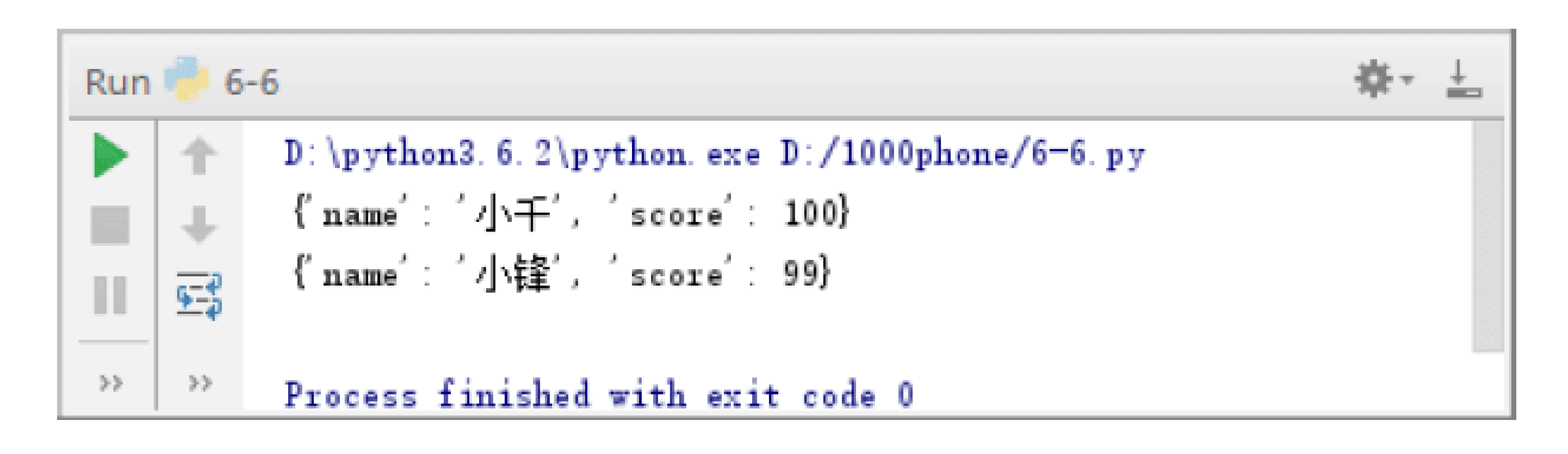


图 6.7　例 6-6 运行结果

在例 6-6 中，第 3 行与第 4 行通过键修改所对应的值。从运行结果可发现，修改后字典中的元素发生了变化。

6.3.4　添加元素

通过键修改值时，如果键不存在，则会在字典中添加该键值对，如例 6-7 所示。

例 6-7　向字典中添加元素。

```
std = {'name':'小千', 'score':100}
```

```
2    std['name'] = '小锋'      # 该键存在,修改键对应的值
3    std['age'] = 18           # 该键不存在,添加该键值对
4    print(std)
```

运行结果如图 6.8 所示。

```
Run  6-7
D:\python3.6.2\python.exe D:/1000phone/6-7.py
{'name': '小锋', 'score': 100, 'age': 18}

Process finished with exit code 0
```

图 6.8　例 6-7 运行结果

在例 6-7 中，第 2 行修改键'name'所对应的值为'小锋'，第 3 行将键值对（'age':18）添加到字典中。

此外，还可以通过 update()函数修改某键对应的值或添加元素，如例 6-8 所示。

例 6-8　update()函数的用法。

```
1    std = {'name':'小千', 'score':100}
2    new = {'name':'小锋'}
3    std.update(new)  # 修改键所对应的值
4    print(std)
5    add = {'age':18}
6    std.update(add)  # 添加元素
7    print(std)
```

运行结果如图 6.9 所示。

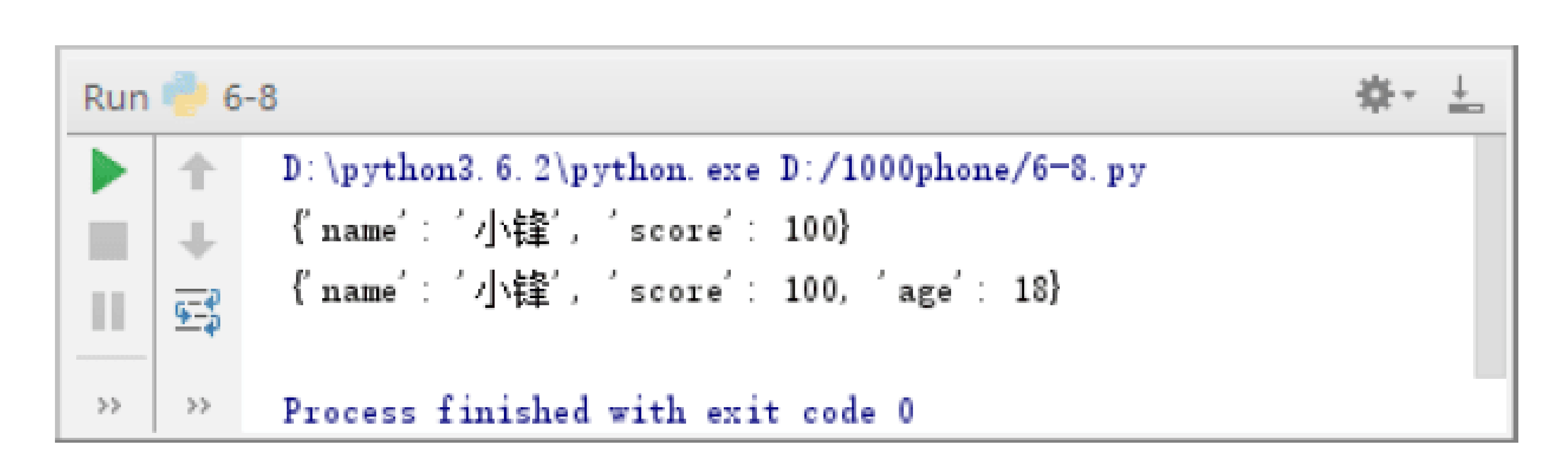


图 6.9　例 6-8 运行结果

在例 6-8 中，第 3 行修改键'name'所对应的值为'小锋'，第 6 行将键值对（'age':18）添加到字典 std 中。

6.3.5　删除元素

删除字典中的元素可以通过“del 字典名[键]”实现，如例 6-9 所示。

例 6-9　删除字典中的元素。

```
std = {'name':'小千', 'score':100}
del std['score']
```

```
3 print(std)
```

运行结果如图 6.10 所示。

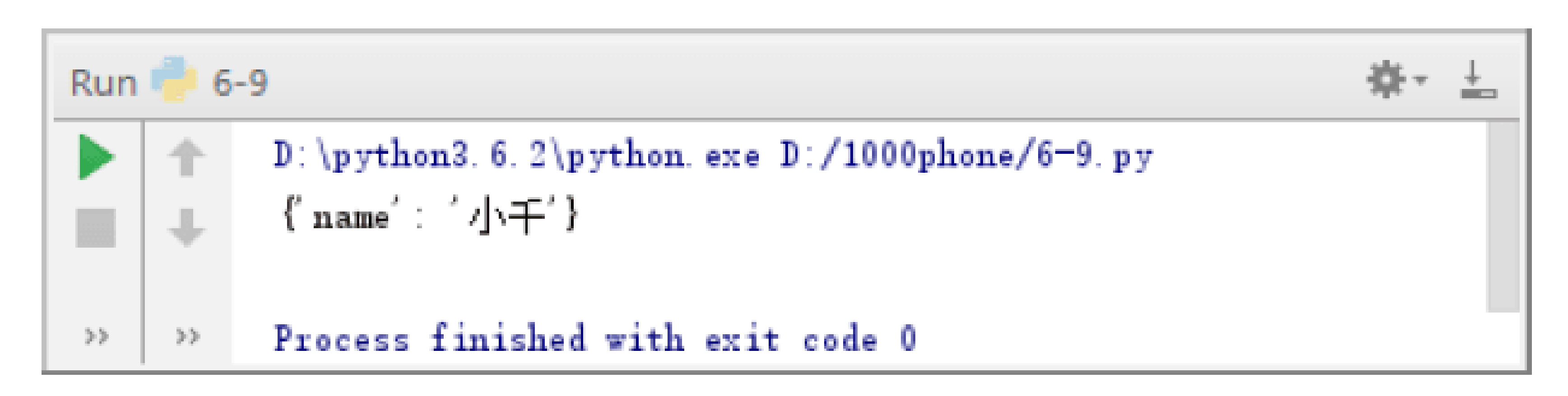


图 6.10 例 6-9 运行结果

在例 6-9 中，第 2 行通过 del 删除字典中的键值对（'score':100）。

如果想删除字典中所有元素，则可以使用 clear()实现，如例 6-10 所示。

例 6-10 删除字典中的所有元素。

```
1 std = {'name': '小千', 'score': 100}
2 std.clear()
3 print(std)
```

运行结果如图 6.11 所示。

```
Run 6-10
D:\python3.6.2\python.exe D:/1000phone/6-10.py
{}

Process finished with exit code 0
```

图 6.11 例 6-10 运行结果

在例 6-10 中，第 2 行通过 clear()删除字典中所有的元素，此时该字典是一个空字典。

注意使用“del 字典名”可以删除字典，删除后，字典就完全不存在了，如例 6-11 所示。

例 6-11 通过 del 删除字典。

```
1 std = {'name': '小千', 'score': 100}
2 del std
3 print(std)
```

运行结果如图 6.12 所示。

```
Run 6-11
D:\python3.6.2\python.exe D:/1000phone/6-11.py
Traceback (most recent call last):
  File "D:/1000phone/6-11.py", line 3, in <module>
    print(std)
NameError: name 'std' is not defined

Process finished with exit code 1
```

图 6.12 例 6-11 运行结果

在例 6-11 中，第 2 行通过 del 删除字典，第 3 行试图访问删除后的字典，程序会提示 std 未定义。

6.3.6 复制字典

有时需要将字典复制一份以便用于其他操作，这样原字典数据不受影响，这时可以通过 copy()函数来实现，如例 6-12 所示。

例 6-12 复制字典。

```
1  std = {'name':'小千', 'score':100}
2  s = std.copy()
3  del s['score']
4  print(s)
5  print(std)
```

运行结果如图 6.13 所示。

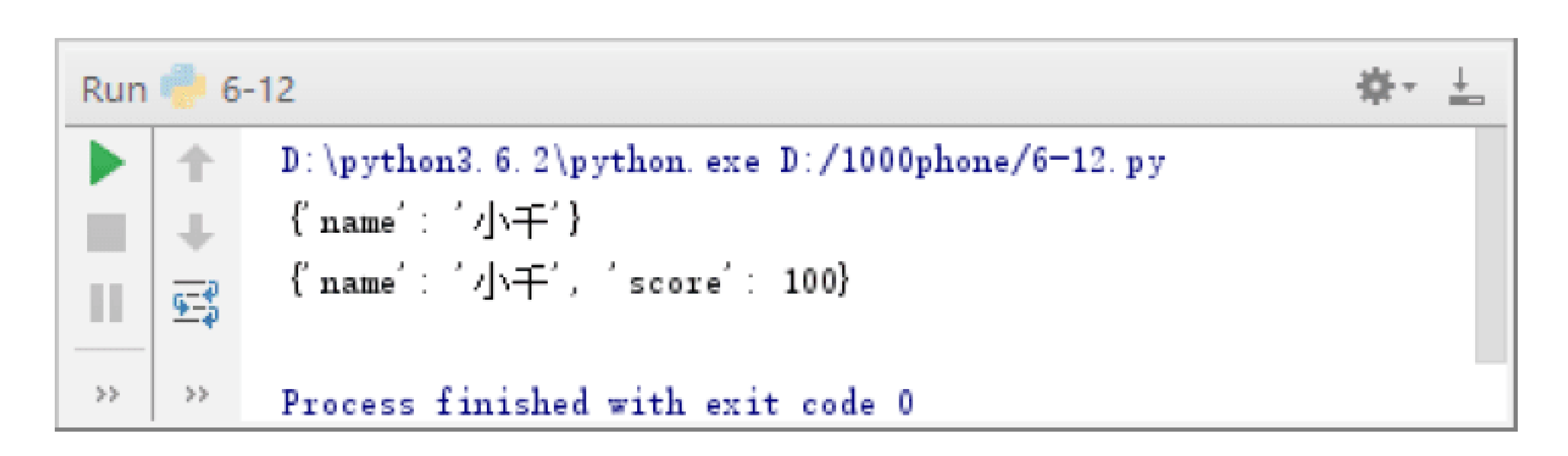


图 6.13 例 6-12 运行结果

在例 6-12 中，第 2 行通过 copy()将字典 std 中数据复制一份赋值给字典 s。第 3 行删除字典 s 中元素（'score':100）。从运行结果可发现，程序对字典 s 的操作并不会影响字典 std。

6.3.7 成员运算

字典中可以使用成员运算符（in、not in）来判断某键是否在字典中，如例 6-13 所示。

例 6-13 字典中有关成员运算符的使用。

```
1  std = {'name':'小千', 'score':100}
2  print('name' in std)
3  print('score' not in std)
```

运行结果如图 6.14 所示。

在例 6-13 中，第 2 行与第 3 行通过成员运算符判断键是否在字典中。注意该运算符只能判断键是否在字典中，不能判断值是否在字典中。

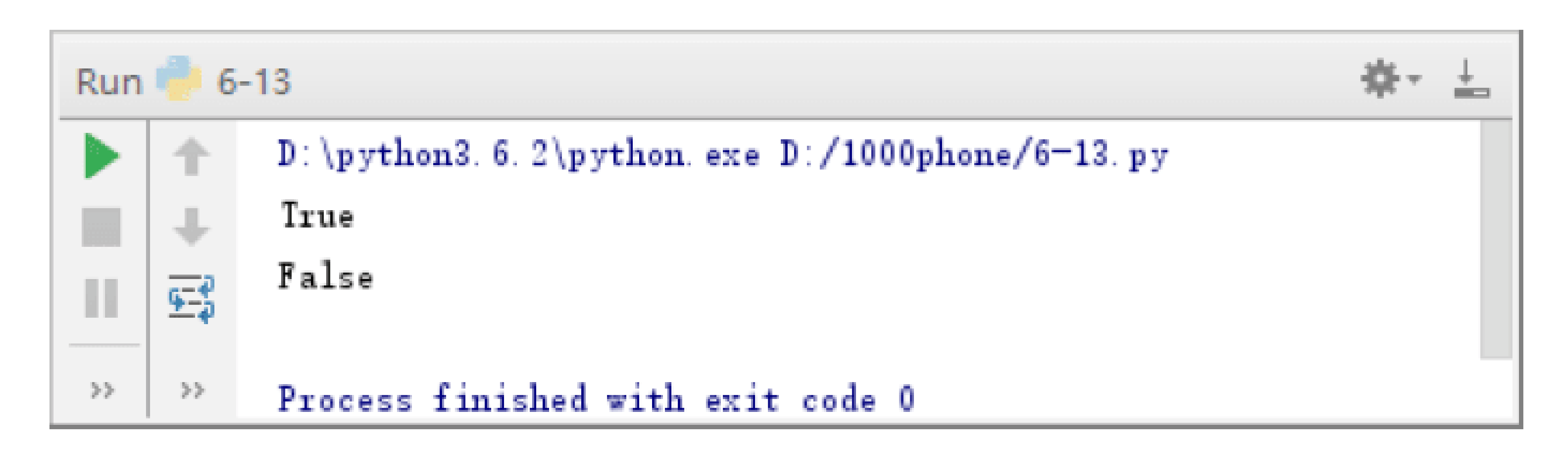


图 6.14　例 6-13 运行结果

6.3.8　设置默认键值对

有时需要为字典中某个键设置一个默认值，则可以使用 setdefault()函数，如例 6-14 所示。

例 6-14　设置默认键值对。

```
1  std = {'name':'小千', 'score':100}
2  name = std.setdefault('school', '千锋教育')
3  print(name, std)
4  name = std.setdefault('school', '扣丁学堂')
5  print(name, std)
```

运行结果如图 6.15 所示。

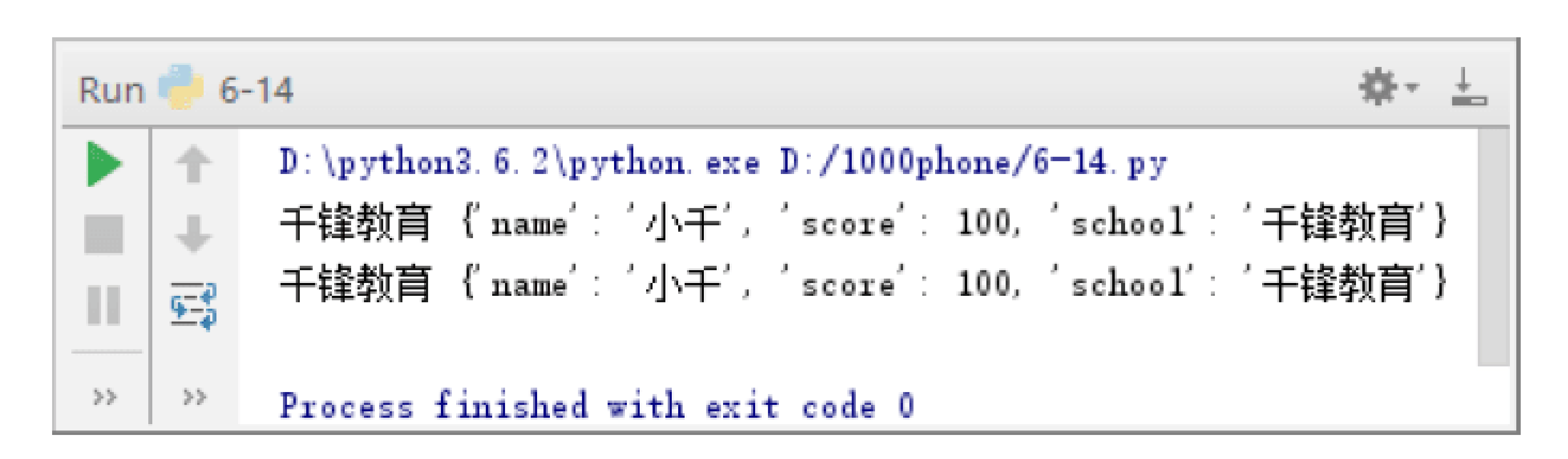


图 6.15　例 6-14 运行结果

在例 6-14 中，程序执行第 2 行时，键'school'不在字典中，此时 setdefault()函数向字典中加入键值对'school': '千锋教育'，并将'千锋教育'作为返回值赋值给 name。程序执行第 4 行时，键'school'已在字典中，此时 setdefault()函数只将该键对应的值'千锋教育'作为返回值赋值给 name。

6.3.9　获取字典中的所有键

keys()函数可以获取字典中所有元素的键，如例 6-15 所示。

例 6-15　获取字典中所有元素的键。

```
std = {'name': '小千', 'score': 100}
```

```
2    print(std.keys())
3    for key in std.keys():
4        print(key)
```

运行结果如图 6.16 所示。

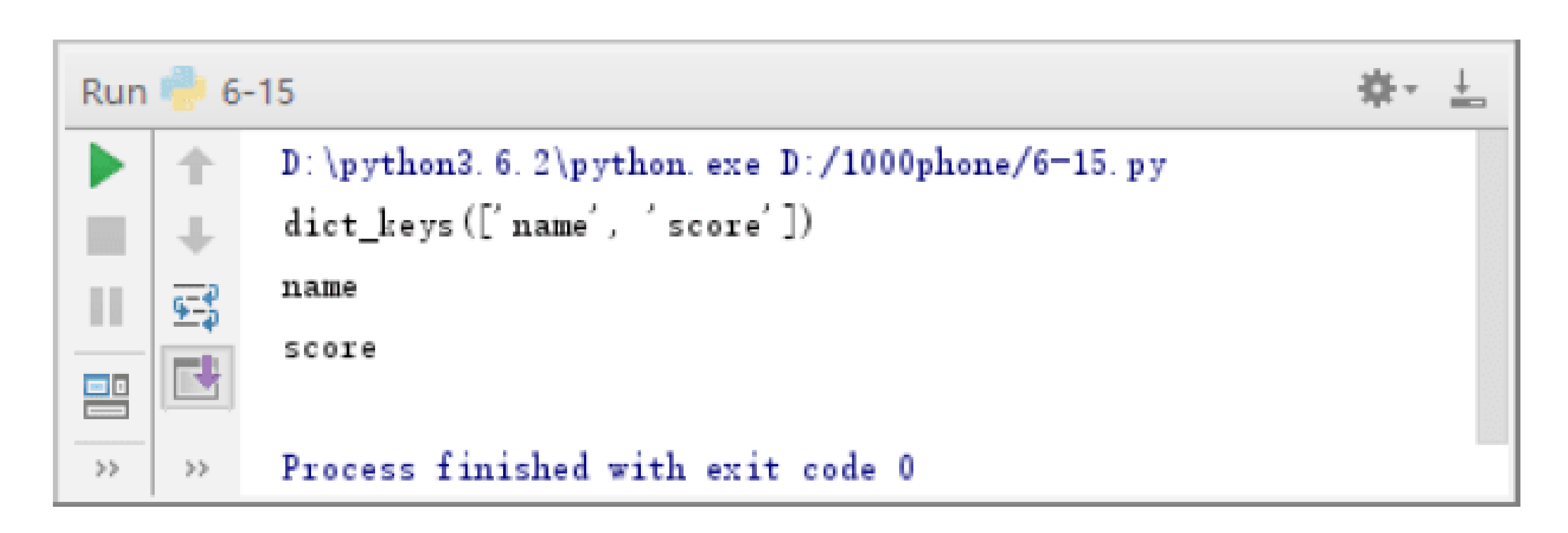


图 6.16　例 6-15 运行结果

在例 6-15 中，第 2 行打印 keys()函数的返回值，第 3、4 行通过 for 循环遍历 keys()函数返回值并打印每一项。

6.3.10　获取字典中的所有值

values()函数可以获取字典中所有元素键所对应的值，如例 6-16 所示。

例 6-16　获取字典中所有元素键所对应的值。

```
1    std = {'name': '小千', 'score': 100}
2    print(std.values())
3    for value in std.values():
4        print(value)
```

运行结果如图 6.17 所示。

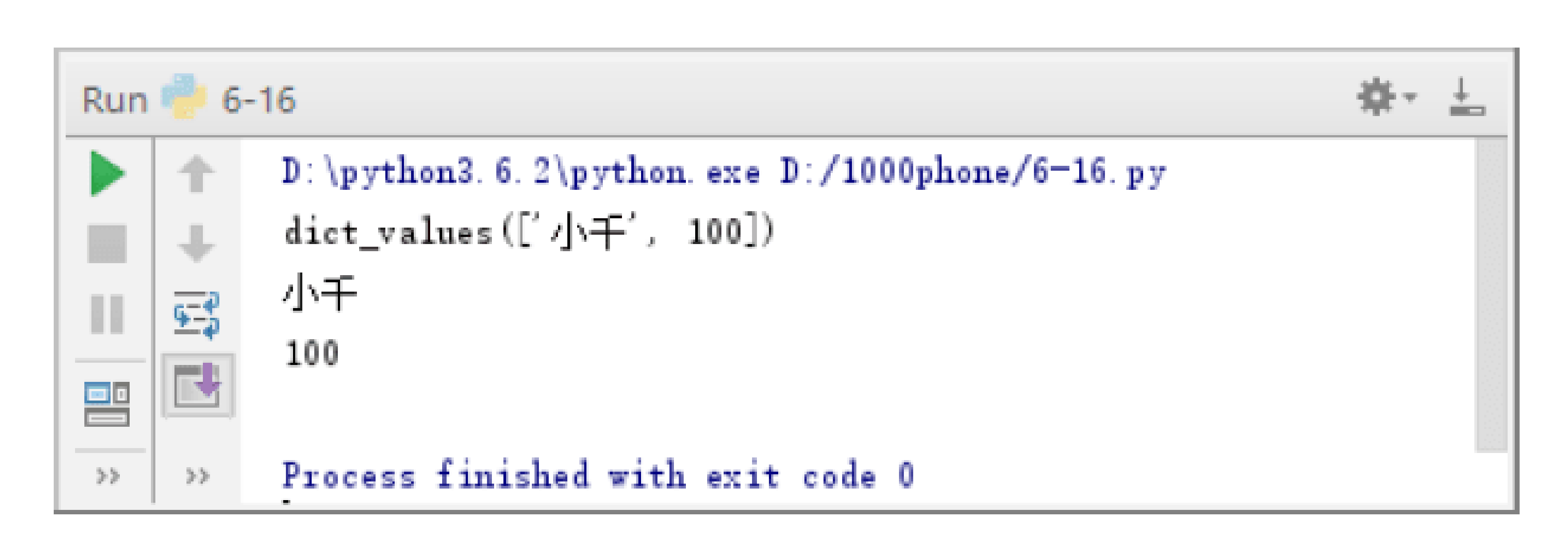


图 6.17　例 6-16 运行结果

在例 6-16 中，第 2 行打印 values()函数的返回值，第 3、4 行通过 for 循环遍历 values()函数返回值并打印每一项。

6.3.11　获取字典中所有的键值对

items()函数可以获取字典中所有的键值对，如例 6-17 所示。

例 6-17 获取字典中所有的键值对。

```
1    std = {'name': '小千', 'score': 100}
2    print(std.items())
3    for item in std.items():
4        print(item)
```

运行结果如图 6.18 所示。

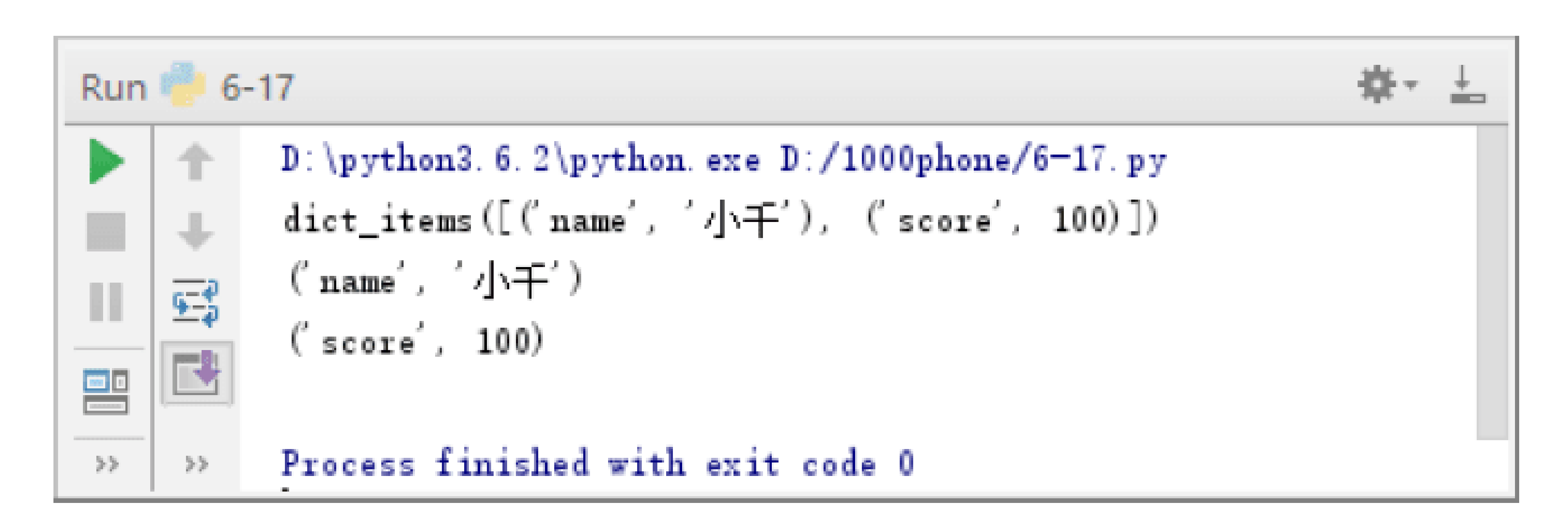


图 6.18 例 6-17 运行结果

在例 6-17 中，第 2 行打印 items()函数的返回值，第 3、4 行通过 for 循环遍历 items()函数返回值并打印每一项。从运行结果可看出，每一项都是由键与值组成的元组。

此外，items()函数与 for 循环结合可以遍历字典中的键值对，如例 6-18 所示。

例 6-18 遍历字典中的键值对。

```
1    std = {'name': '小千', 'score': 100}
2    for key, value in std.items():
3        print('key = %s, value = %s'%(key, value))
```

运行结果如图 6.19 所示。

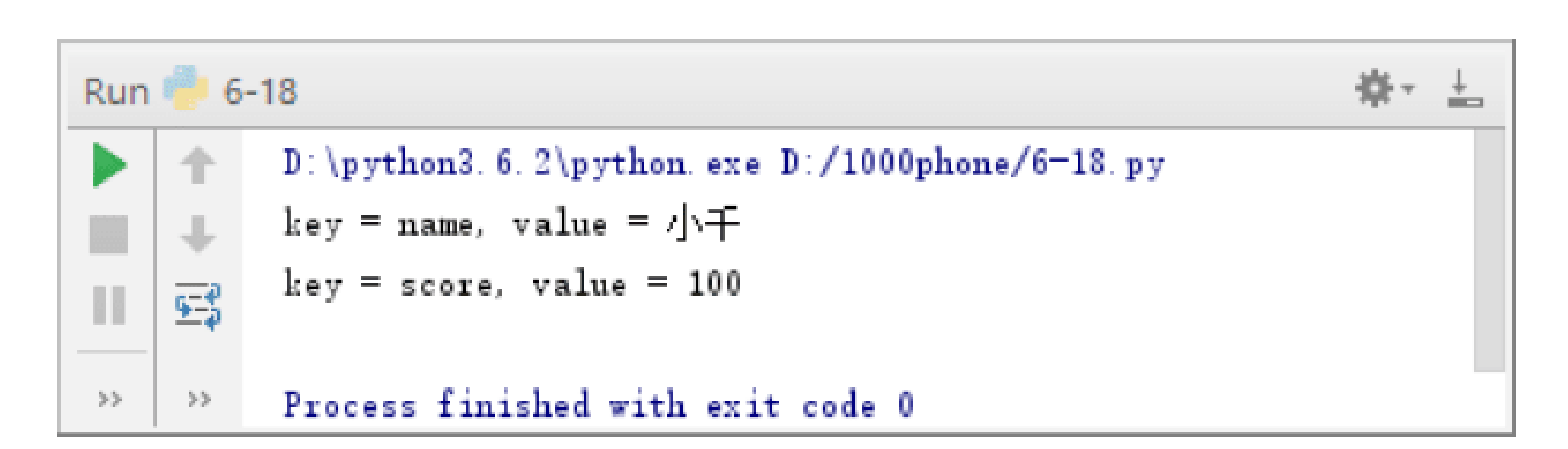


图 6.19 例 6-18 运行结果

在例 6-18 中，第 2、3 行通过 for 循环遍历 items()函数返回值并将每一项中的键与值分别赋值给 key 与 value。

6.3.12 随机删除元素

popitem()函数可以随机返回并删除一个元素，如例 6-19 所示。

例 6-19 随机返回并删除一个元素。

```
std = {'name': '小千', 'score': 100, 'school':'千锋教育'}
```

```
2    item = std.popitem()
3    print(item, std)
```

运行结果如图 6.20 所示。

```
Run 6-19
D:\python3.6.2\python.exe D:/1000phone/6-19.py
('school', '千锋教育') {'name': '小千', 'score': 100}

Process finished with exit code 0
```

图 6.20　例 6-19 运行结果

在例 6-19 中，第 2 行执行 popitem()函数后，删除字典中最后一个元素。注意该函数返回一个元组。

此外，pop()函数可以根据指定的键删除元素，如例 6-20 所示。

例 6-20　根据指定的键删除元素。

```
1    std = {'name': '小千', 'score': 100, 'school':'千锋教育'}
2    item = std.pop('score')
3    print(item, std)
```

运行结果如图 6.21 所示。

```
Run 6-20
D:\python3.6.2\python.exe D:/1000phone/6-20.py
100 {'name': '小千', 'school': '千锋教育'}

Process finished with exit code 0
```

图 6.21　例 6-20 运行结果

在例 6-20 中，第 2 行执行 pop()函数后，删除字典中键为'score'的元素。注意该函数返回键所对应的值，而不是键值对。

6.4　集合的概念

集合是由一组无序排列且不重复的元素组成的，具体示例如下：

```
set1 = {1, 2, 'a'}
```

集合使用大括号表示，元素类型可以是数字类型、字符串、元组，但不可以是列表、字典，具体示例如下：

```
set2 = {2 , ['a', 1]}    # 错误,元素包含列表
set3 = {2 , {'a':1}}     # 错误,元素包含字典
set3 = {2 , ('a', 1)}    # 正确,元素包含元组
```

使用大括号创建的集合属于可变集合，即可以添加或删除元素。此外，还存在一种不可变集合，即不允许添加或删除元素。

接下来演示创建这两种集合的方法，如例 6-21 所示。

例 6-21 创建集合的方法。

```
1  set1 = set('xiaoqian')              # 通过 set()创建可变集合
2  print(type(set1), set1)
3  set2 = set(('xiaoqian', 'xiaofeng'))
4  set3 = set(['xiaoqian', 'xiaofeng'])
5  print(set2, set3)
6  fset1 = frozenset('xiaofeng')  # 通过 frozenset()创建不可变集合
7  print(type(fset1))
8  print(fset1)
```

运行结果如图 6.22 所示。

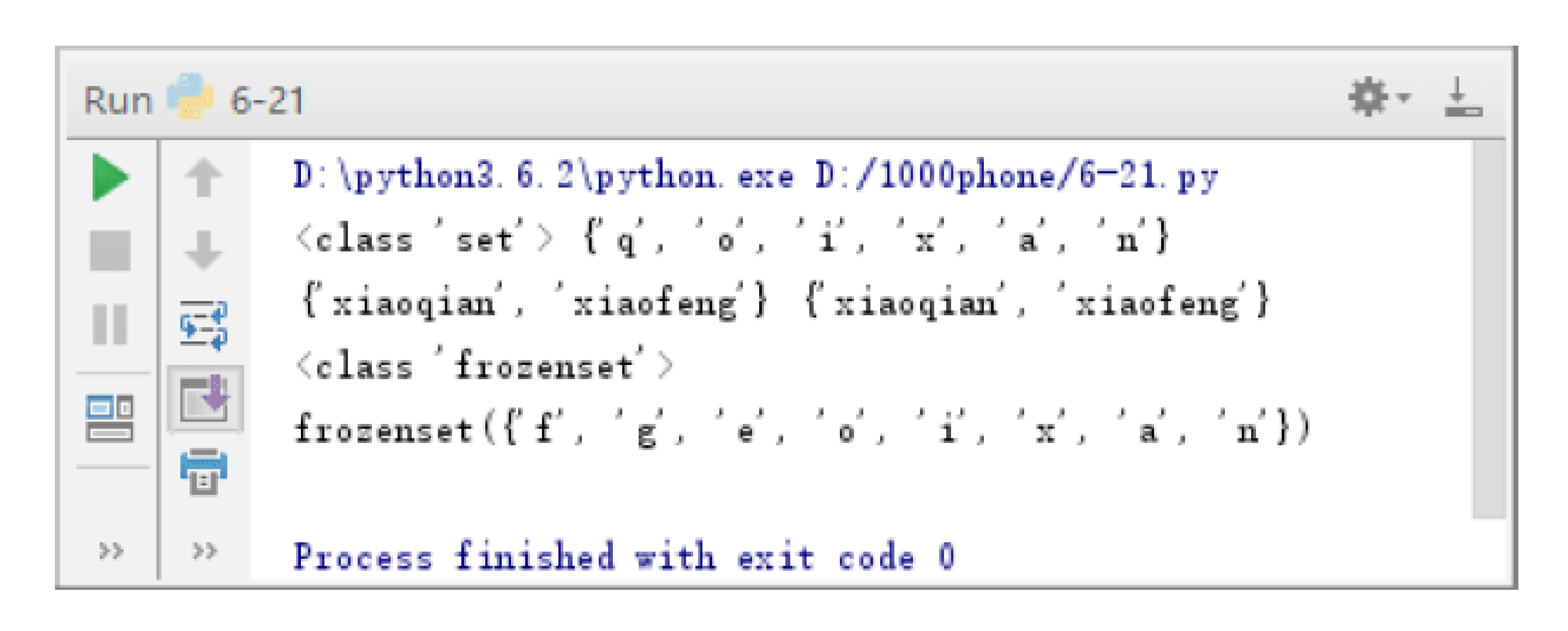


图 6.22 例 6-21 运行结果

在例 6-21 中，第 1 行通过 set()函数创建可变集合并将字符串中去重后的字符作为集合的元素。第 3 行将元组作为 set()函数的参数创建集合 set2。第 4 行将列表作为 set()函数的参数创建集合 set3。第 6 行通过 frozenset ()函数创建不可变集合。

集合的一个重要用途是将一些数据结构中的重复元素去除，如例 6-22 所示。

例 6-22 集合的用途。

```
list1 = [1, 2, 3, 4, 3, 2, 1]
set1 = set(list1)    # 将列表转换为集合并去重
list2 = list(set1)   # 将集合转换为列表
print(list2)
```

运行结果如图 6.23 所示。

```
Run 6-22
D:\python3.6.2\python.exe D:/1000phone/6-22.py
[1, 2, 3, 4]

Process finished with exit code 0
```

图 6.23 例 6-22 运行结果

在例 6-22 中，第 2 行通过 set()函数将列表转换为集合，集合中的元素是不重复的。第 3 行通过 list()函数将集合转换为列表，此时列表中的元素也是不重复的。

6.5 集合的常用操作

同其他数据类型类似，集合也有一系列常用的操作，例如添加元素、删除元素等。通过这些操作，可以很方便地处理集合。

6.5.1 添加元素

集合中添加元素可以使用 add()和 update()函数，如例 6-23 所示。

例 6-23 向集合中添加元素。

```
1  set1, set2 = {1, 2, 3}, {3, 4, 5, 6}
2  set1.add(4)
3  print(set1)
4  set1.update(set2)
5  print(set1)
```

运行结果如图 6.24 所示。

```
Run 6-23
D:\python3.6.2\python.exe D:/1000phone/6-23.py
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6}

Process finished with exit code 0
```

图 6.24 例 6-23 运行结果

在例 6-23 中，第 2 行通过 add()函数将元素 4 添加到集合 set1，第 4 行通过 update()函数将集合 set2 中的元素添加到集合 set1。

6.5.2 删除元素

在集合中删除元素可以使用 remove()和 discard()函数，如例 6-24 所示。

例 6-24 在集合中删除元素。

```
set1  = {1, 2, 3, 4}
set1.remove(3)   # 删除集合 set1 中元素 3,remove()删除不存在元素时会报错
set1.discard(4)  # 删除集合 set1 中元素 4,discard()删除不存在元素时不会报错
set1.discard(5)
```

```
print(set1)
set1.clear()      # 清空集合
print(set1)
```

运行结果如图 6.25 所示。

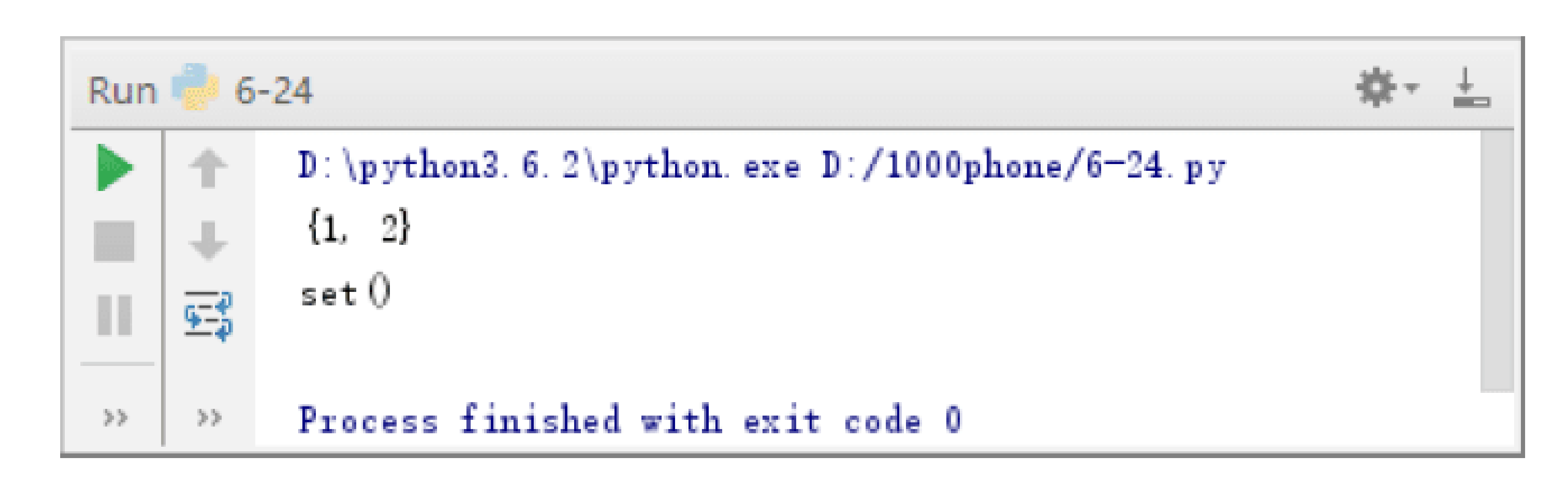


图 6.25　例 6-24 运行结果

在例 6-24 中，应注意 remove()和 discard()函数的区别。

6.5.3　集合运算

集合可以参与多种运算，如表 6.1 所示。

表 6.1　集合中的运算

<table>
<tr><th>运　　算</th><th>说　　明</th><th>运　　算</th><th>说　　明</th></tr>
<tr><td>x in set1</td><td>检测 x 是否在集合 set1 中</td><td>set1 | set2</td><td>并集</td></tr>
<tr><td>set1 = = set2</td><td>判断集合是否相等</td><td>set1 & set2</td><td>交集</td></tr>
<tr><td>set1 <= set2</td><td>判断 set1 是否是 set2 的子集</td><td>set1 – set2</td><td>差集</td></tr>
<tr><td>set1 < set2</td><td>判断 set1 是否是 set2 的真子集</td><td>set1 ^ set2</td><td>对称差集</td></tr>
<tr><td>set1 >= set2</td><td>判断 set1 是否是 set2 的超集</td><td>set1 |= set2</td><td>将 set2 的元素并入 set1</td></tr>
<tr><td>set1 > set2</td><td>判断 set1 是否是 set2 的真超集</td><td></td><td></td></tr>
</table>

接下来演示这些运算的用法，如例 6-25 所示。

例 6-25　集合中的运算。

```
set1, set2 = {1, 2, 3}, {2, 3, 4}
print(1 in set1)     # set1 中包含元素 1
print(set1 == set2)  # set1 与 set2 不相等
print(set1 > set2)   # set1 不是 set2 的真超集
print(set1 >= set2)  # set1 不是 set2 的超集
print(set1 | set2)   # 并集
print(set1 & set2)   # 交集
print(set1 - set2)   # 差集
print(set1 ^ set2)   # 对称差集
set1 |= set2         # 将 set2 并入 set1
print(set1)
```

运行结果如图 6.26 所示。

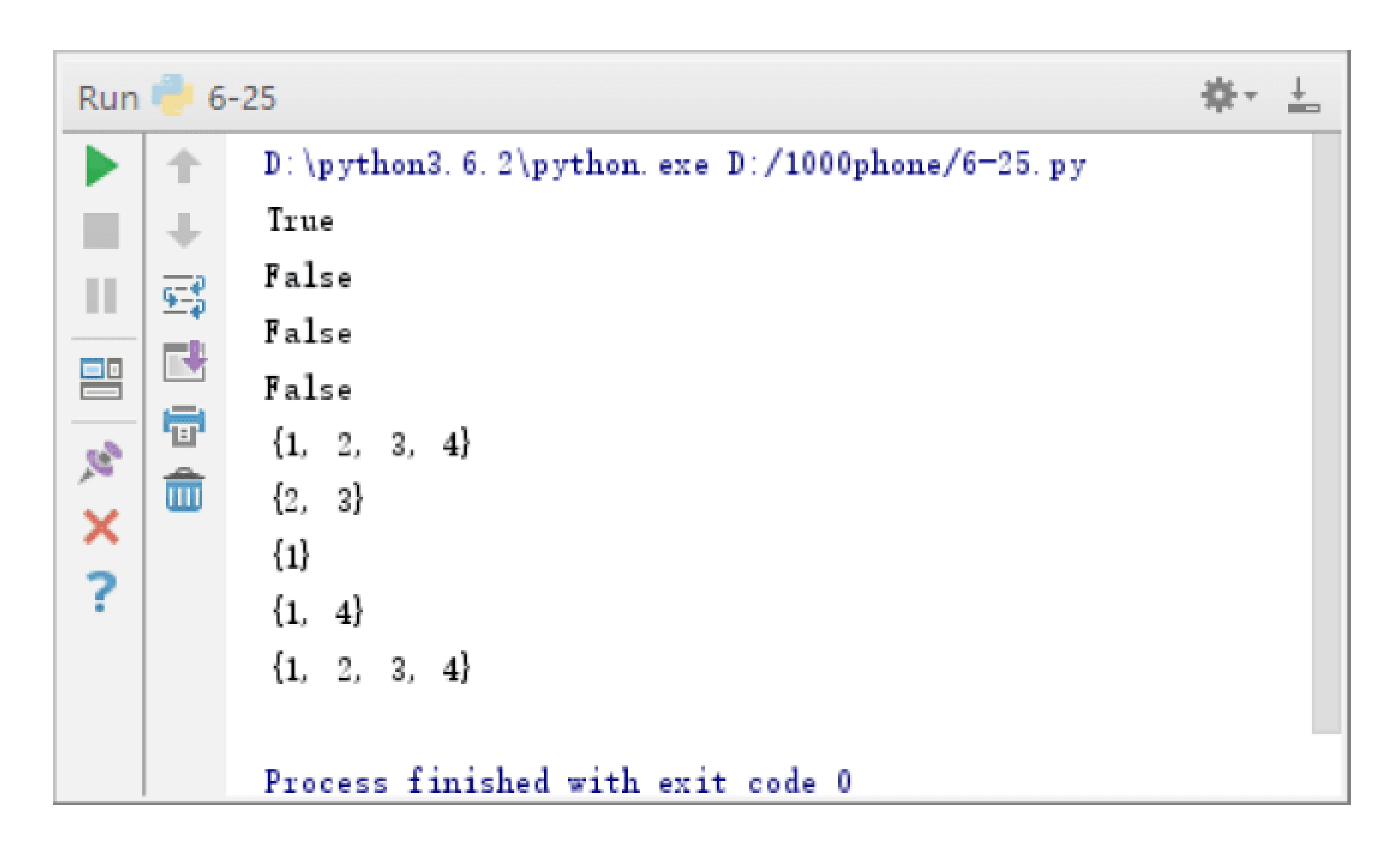


图 6.26　例 6-25 运行结果

在例 6-25 中，除了“set1 |= set2”外，所有的运算都不会影响 set1 与 set2 中的元素。

除了上述运算符外，还可以通过 union()、intersection()与 difference()函数实现集合的并集、交集与差集，如例 6-26 所示。

例 6-26　集合的并集、交集与差集。

```
set1, set2 = {1, 2, 3}, {2, 3, 4}
print(set1.union(set2))          # 并集
print(set1.intersection(set2))   # 交集
print(set1.difference(set2))     # 差集
```

运行结果如图 6.27 所示。

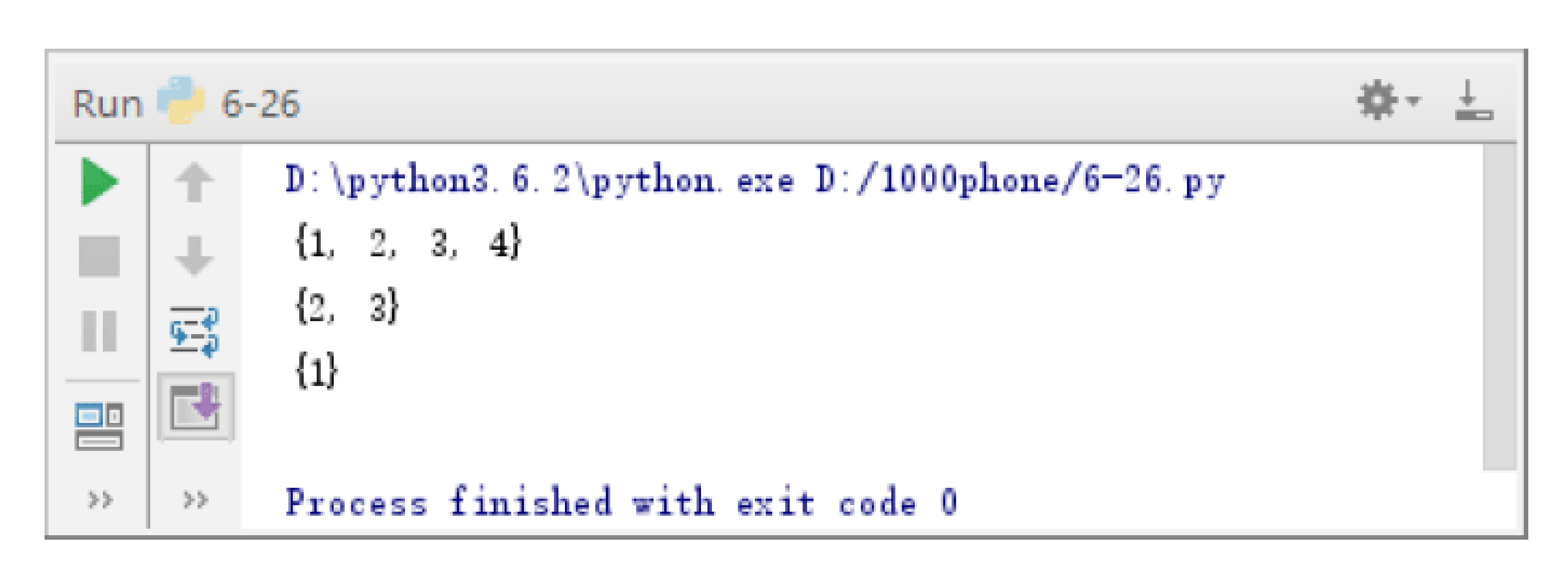


图 6.27　例 6-26 运行结果

在例 6-26 中，这 3 个函数的调用都不会影响 set1 与 set2 中的元素。

6.5.4　集合遍历

可以通过 for 循环遍历集合中的元素，如例 6-27 所示。

例 6-27　集合的遍历。

```
set1 = {1, 2, 3, 4}
for num in set1:
```

```
    print(num, end = ' ')
```

运行结果如图 6.28 所示。

```
Run 6-27
D:\python3.6.2\python.exe D:/1000phone/6-27.py
1 2 3 4
Process finished with exit code 0
```

图 6.28 例 6-27 运行结果

6.6 字典推导与集合推导

字典推导与列表推导相似，它将推导出一个字典，具体示例如下：

```
dict1 = {x : x * x for x in range(5)}
```

字典推导使用大括号包围，并且需要两个表达式：一个生成 key，一个生成 value，两个表达式之间使用冒号分隔，结果返回字典。若通过 print()打印 dict1，则输出结果为：

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

上述就是一个简单的字典推导，接下来演示稍微复杂的字典推导，如例 6-28 所示。

例 6-28 字典推导。

```
1   dict1 = [(86, 'china'), (91, 'india'), (1, 'united states')]
2   dict2 = {country: code for code, country in dict1}
3   print(dict2)
4   print({code: country for country, code in dict2.items() if code < 90})
```

运行结果如图 6.29 所示。

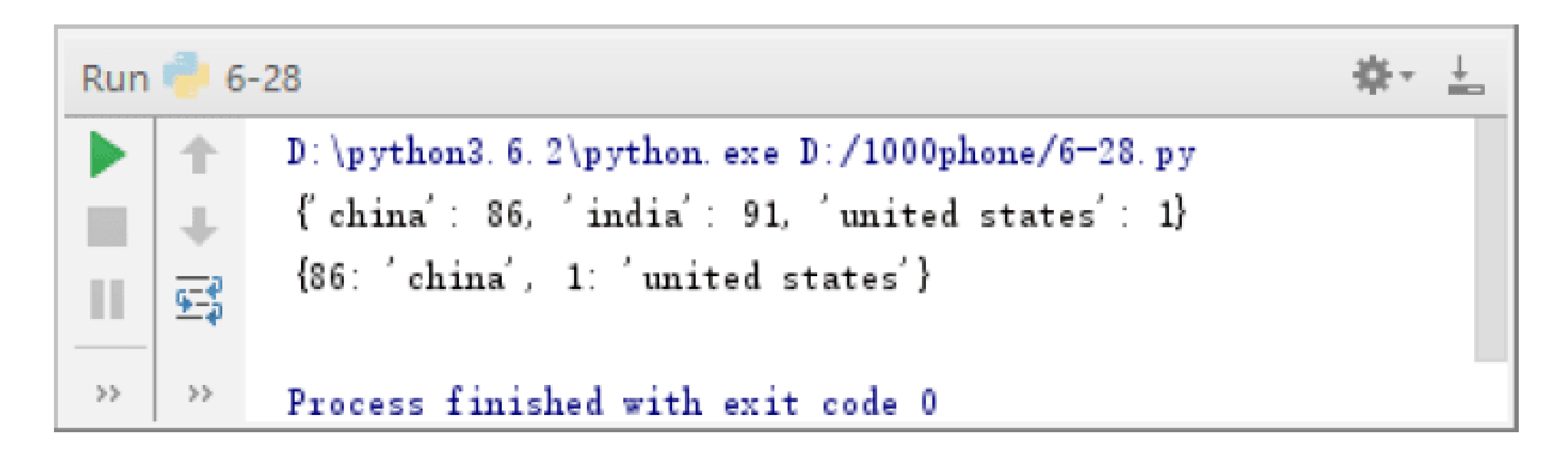


图 6.29 例 6-28 运行结果

在例 6-28 中，第 2 行通过字典推导将 dict1 中的值与键作为 dict2 中的键与值，第 4 行通过字典推导筛选键值小于 90 的元素并返回一个新字典。

集合推导也与列表推导相似，只需将中括号改为大括号，具体示例如下：

```
set1 = {x * x for x in range(5)}
```

集合推导将返回一个集合。若通过 print()打印 set1，则输出结果为：

```
{0, 1, 4, 9, 16}
```

接下来演示集合推导的用法，如例 6-29 所示。

例 6-29 集合推导。

```
1  strings = ['Python', 'HTML', 'PHP', 'VR', 'Java', 'C++']
2  print ({len(s) for s in strings})
```

运行结果如图 6.30 所示。

```
Run 6-29
D:\python3.6.2\python.exe D:/1000phone/6-29.py
{2, 3, 4, 6}

Process finished with exit code 0
```

图 6.30 例 6-29 运行结果

在例 6-29 中，第 2 行使用集合推导创建一个字符串长度的集合，字符串长度相同的值只会在集合中出现一次。

6.7 小 案 例

6.7.1 案例一

小千、小锋与小明在扣丁学堂学习几门不同的 IT 课程，每人已经学习的课时数不同，现用字典保存每人学习的课程与课时数，统计 Python 课程的总课时数，具体实现如例 6-30 所示。

例 6-30 统计 Python 课程的总课时数。

```
std = {
    '小千':{'Python':10, 'PHP':5},
    '小锋':{'Python':8, 'UI':'4'},
    '小明':{'Python':2, 'UI':5, 'PHP':'1'},
}
num = 0
for value in std.values():
    num += value.get('Python', 0) # 若不存在'Python',则返回 0
print('Python 课程总课时数为%d'%num)
```

运行结果如图 6.31 所示。

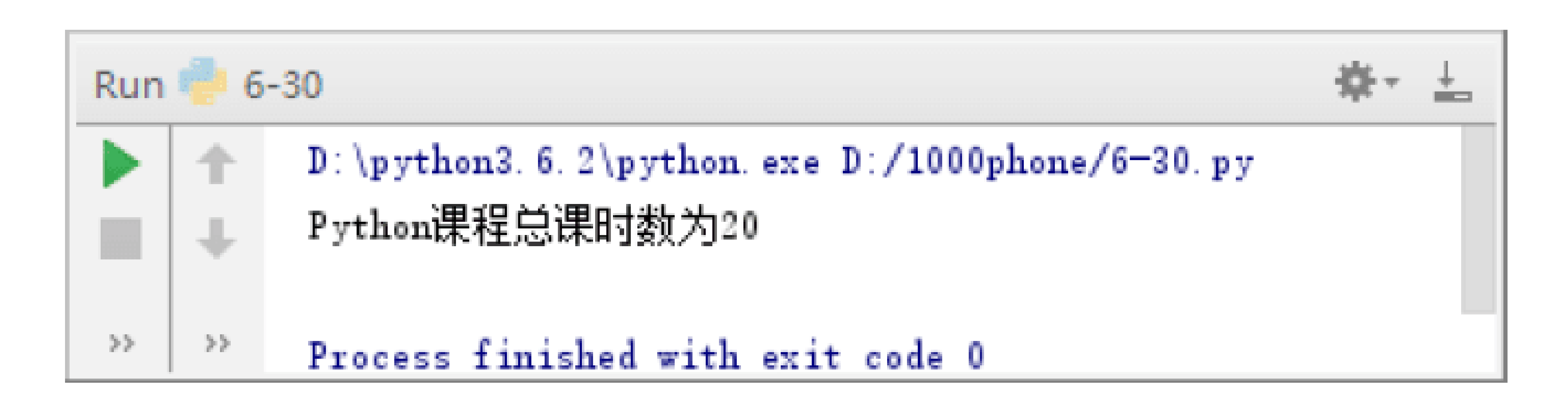


图 6.31　例 6-30 运行结果

在例 6-30 中，程序使用字典的嵌套保存每人学习的课程与课时数，然后通过 for 循环遍历字典中的值来统计 Python 的总课时数。

6.7.2　案例二

输入一句英文，统计英文中出现的字母及次数，使用字典保存每个字母及次数，具体实现如例 6-31 所示。

例 6-31　统计所输入的英文中出现的字母及次数。

```
1    s = input('请输入一句英文：')
2    s = s.upper()
3    dict1 = {chr(n): 0 for n in range(65, 91)}
4    for char in s:
5        if 'A' <= char <= 'Z':
6            dict1[char] += 1
7    list1 = list(dict1.items())
8    for ele in list1:
9        if ele[1] != 0:
10           print(ele[0] + ':', ele[1])
```

运行结果如图 6.32 所示。

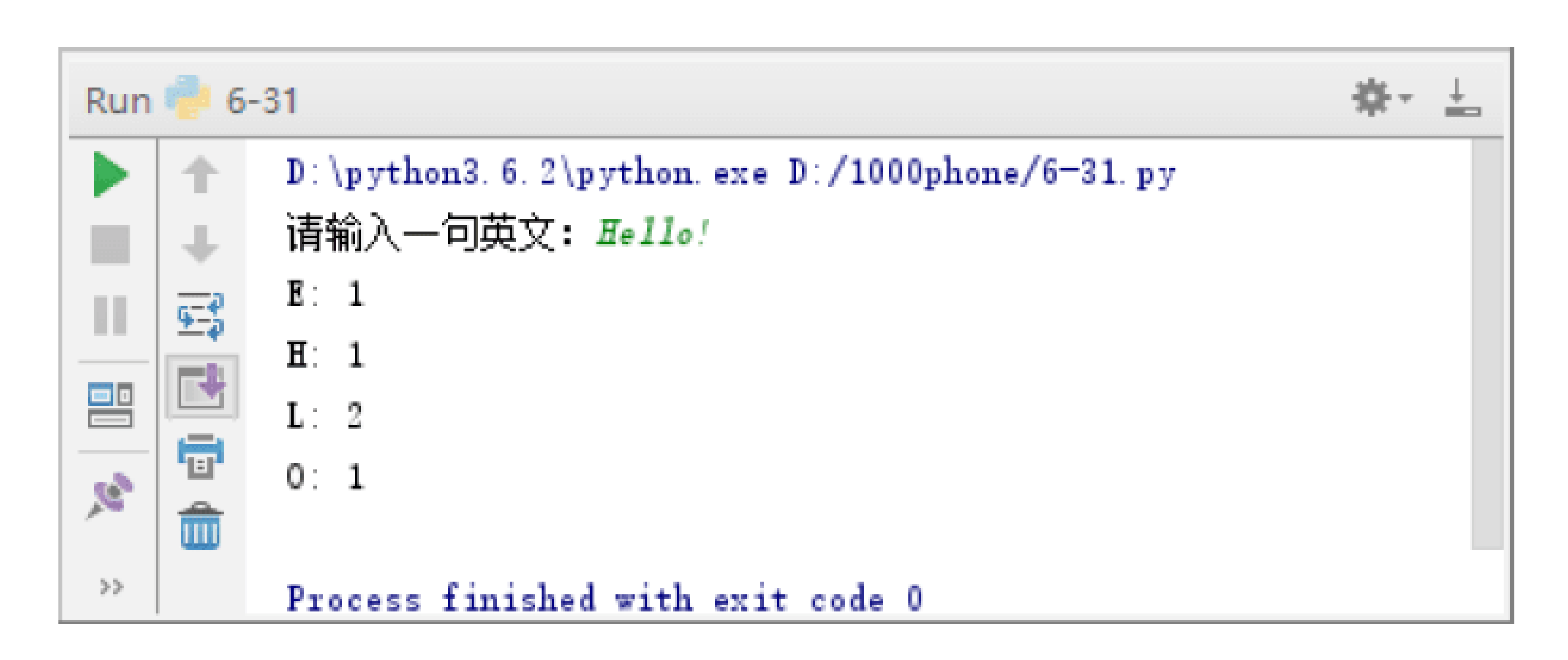


图 6.32　例 6-31 运行结果

在例 6-31 中，第 3 行使用字典推导生成一个字典，其中键为字母，值为 0。第 4 行通过 for 循环遍历输入的字符串，每遍历一个字母相应的字典中对应值加 1。第 7 行将生成一个列表，列表中每个元素为元组。

6.8 本章小结

本章主要介绍了 Python 中的字典与集合，两者都使用大括号表示。字典中每个元素都是由键与值组成的，其中键为不可变类型，而值可以为任意类型。字典在实际开发中经常使用，大家应熟练掌握其常用操作。集合是由一组无序排列且不重复的元素组成的，经常用于去重。集合在实际开发中使用不多，大家只需了解即可。

6.9 习　　题

1．填空题

（1）字典使用________括号表示。
（2）集合使用________括号表示。
（3）字典中每个元素都是由________组成的。
（4）________函数可以获取字典中所有的键值对。
（5）________函数创建一个不可变集合。

2．选择题

（1）下列属于字典的是（　　）。
A．{1:2, 3:4}　　B．[1, 2, 3, 4]　　C．{1, 2, 3, 4}　　D．(1, 2, 3, 4)
（2）下列不可以作为字典键的是（　　）。
A．4　　B．(3, 2, 4, 1)　　C．[4, 1, 3, 2]　　D．'4'
（3）下列可以获取字典中所有值的是（　　）。
A．values()　　B．keys()　　C．get()　　D．getValues()
（4）下列不能使用下标运算的是（　　）。
A．列表　　B．字符串　　C．元组　　D．集合
（5）集合中元素类型不能为（　　）。
A．元组　　B．字符串　　C．数字　　D．字典

3．思考题

（1）简述字典的特征。
（2）简述字典与集合的区别。

4．编程题

由用户输入学生学号与姓名，数据用字典存储，最终输出学生信息（按学号由小到大显示）。



第 7 章 chapter 7

函数(上)

本章学习目标

- 理解函数的概念。
- 掌握函数的定义。
- 掌握函数的参数与返回值。
- 理解变量的作用域。
- 理解函数的嵌套调用与递归调用。

Python 程序是由一系列语句组成的，这些语句都是为了实现某个具体的功能。如果这个功能在整个应用中会经常使用，则每一处需要该功能的位置都要写上同样的代码，这必将会造成大量的冗余代码，不便于开发及后期维护。为此，Python 中引入了函数的概念，它就是为了解决一些常见问题而提前制作的模型。

7.1 函数的概念

函数可以理解为实现某种功能的代码块，这样当程序中需要这个功能时就可以直接调用，而不必每次都编写一次。这就好比生活中使用计算器来计算，当需要计算时，直接使用计算器输入要计算的数，计算完成后显示计算结果，而不必每次计算都通过手写演算出结果。

在程序中，如果需要多次输出“拼搏到无能为力，坚持到感动自己!”，则可以将这个功能写成函数，具体示例如下：

```
def output():
    print('拼搏到无能为力，坚持到感动自己!')
```

当需要使用该函数时，则可以使用以下语句：

```
output()
```

该条语句可以多次使用。函数使减少代码冗余成为现实，并为代码维护节省了不少力气。

Python 中的函数分为内建函数和自定义函数。内建函数是 Python 自带的，即可以直

接使用，如 print()函数、input()函数等。常见的内建函数如表 7.1 所示，本章主要介绍自定义函数。

表 7.1 内建函数

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

7.2 函数的定义

内建函数的数量是有限的，如果大家想自己设计符合使用需求的函数，则可以定义一个函数，其语法格式如下：

```
def 函数名(参数列表):
    函数体
```

在上述语法格式中，需注意以下几点：

- def（即 define，定义）为关键字，表示定义一个函数。
- 函数名是一个标识符，注意不能与关键字重名。
- 小括号之间可以用于定义参数，参数是可选的，但小括号必不可少。
- 函数体以冒号起始，并且缩进。
- 函数体的第一行语句可以选择性地使用文档字符串用来存放函数说明。
- return [表达式]结束函数，将表达式的值返回给调用者，也可以省略。

接下来演示一个简单的自定义函数，如例 7-1 所示。

例 7-1 自定义函数。

```
def sum2num(a, b):
    '''
    求两个数的和
    param a: 左操作数
```

```
5        param b: 右操作数
6        return: 左操作数与右操作数之和
7        '''
8        return a + b
9    x = sum2num(3, 4)
10   print(x)
11   print(sum2num.__doc__)
```

运行结果如图 7.1 所示。

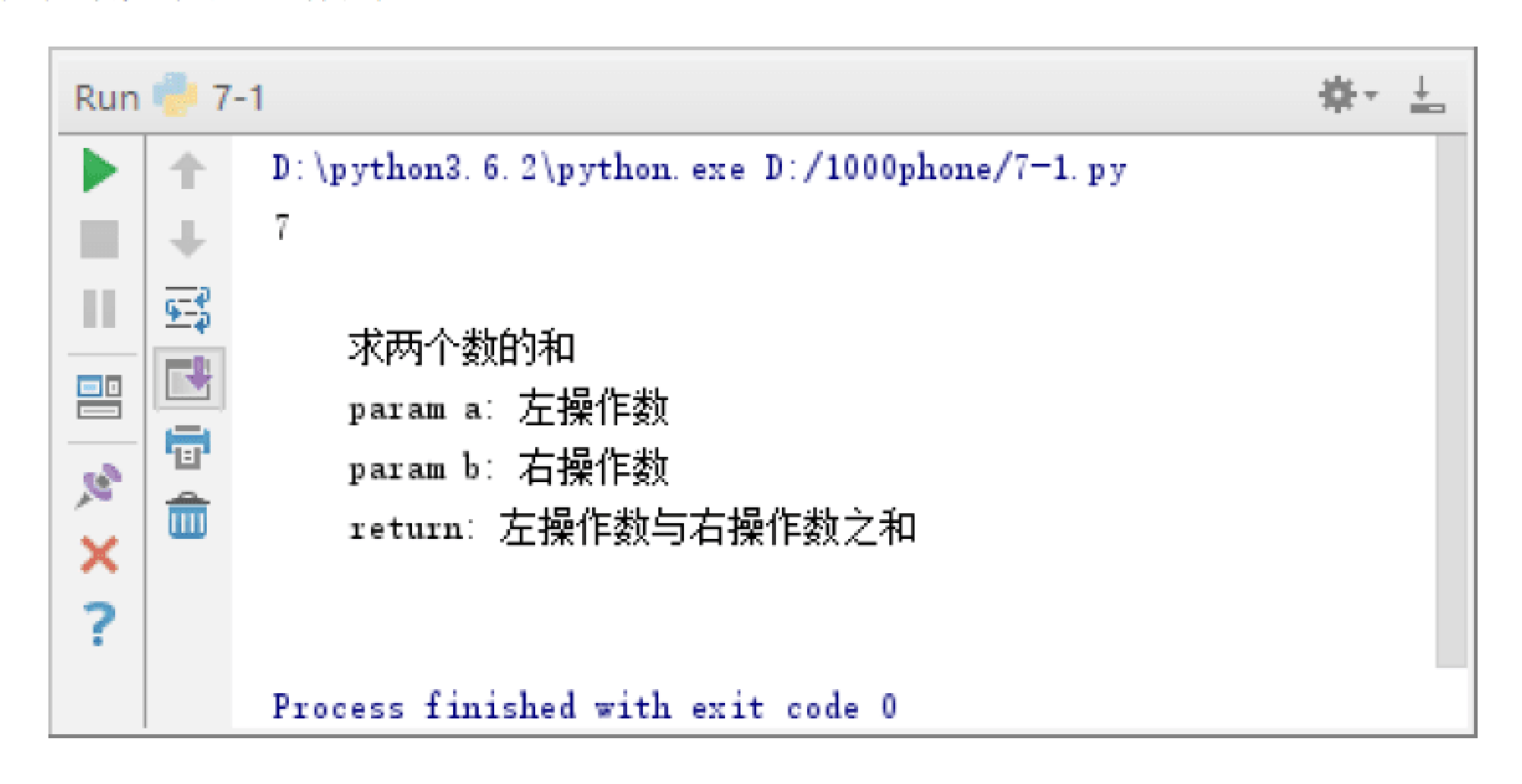


图 7.1　例 7-1 运行结果

在例 7-1 中，第 2～7 行为文档字符串，初学者在初学阶段只需了解即可。若想查看一个函数的文档字符串，则可以通过__doc__属性，如第 11 行所示。关于自定义函数 sum2num()的解释，如图 7.2 所示。

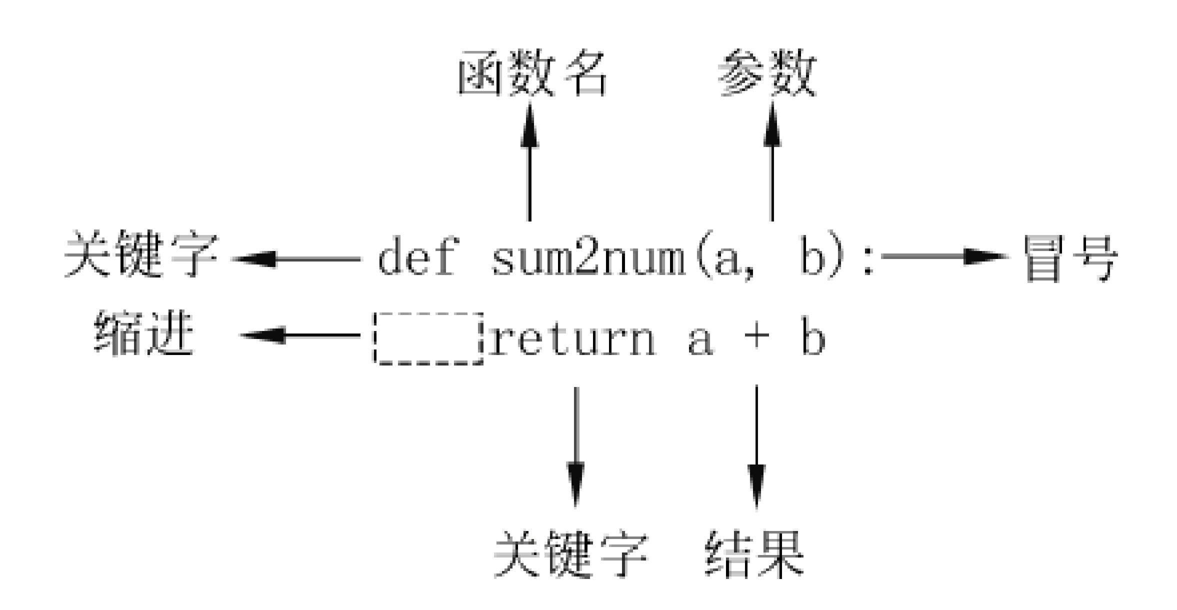


图 7.2　自定义函数

定义函数后，就相当于有了一个具有某些功能的代码。如果想让程序执行这些代码，则需要调用之前定义的函数，其语法格式如下：

```
函数名(参数)
```

在例 7-1 中，求 3 与 4 的和时，则可以通过以下语句实现：

```
sum2num(3, 4)
```

7.3 函数的参数

参数列表由一系列参数组成，并用逗号隔开。在调用函数时，如果需要向函数传递参数，则被传入的参数称为实参，而函数定义时的参数称为形参，实参与形参之间可以传递数据。

7.3.1 位置参数

位置参数是指调用函数时根据函数定义的参数位置来传递函数，如例 7-2 所示。

例 7-2 位置参数的使用。

```
1   def printInfo(name, score):
2       print('姓名：%s\n 成绩：%.2f'%(name, score))
3   printInfo('小千', 98)
4   # printInfo(98, '小千')
```

运行结果如图 7.3 所示。

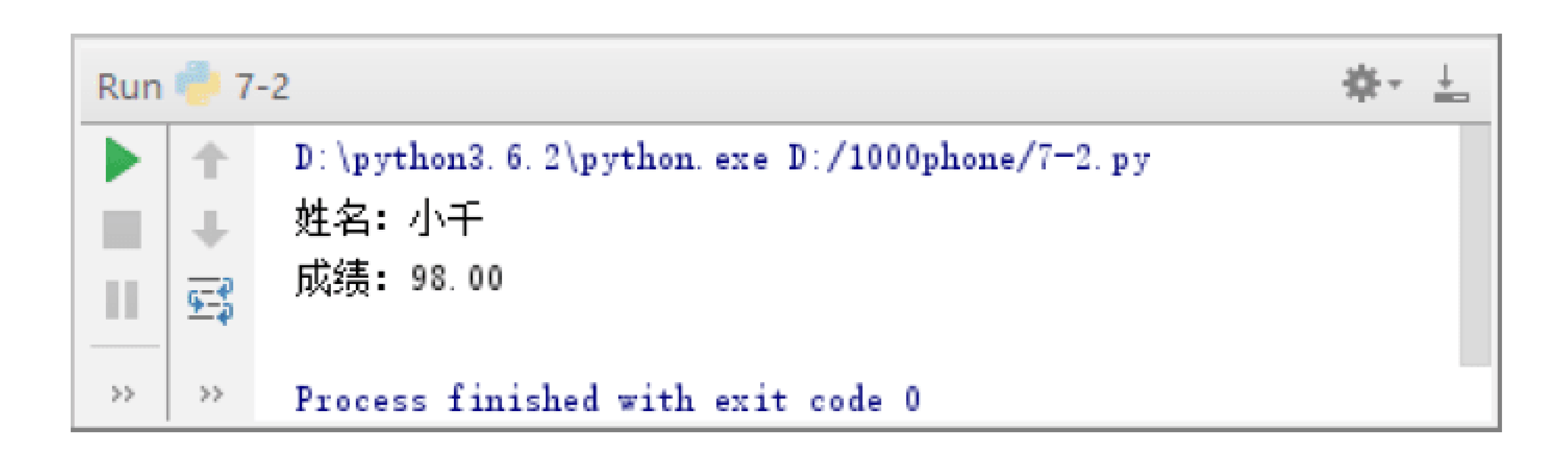


图 7.3 例 7-2 运行结果

在例 7-2 中，第 1、2 行定义 printInfo()函数。第 3 行调用该函数，其数据传递如图 7.4 所示。第 4 行将两个实参的位置调换，则发生错误。

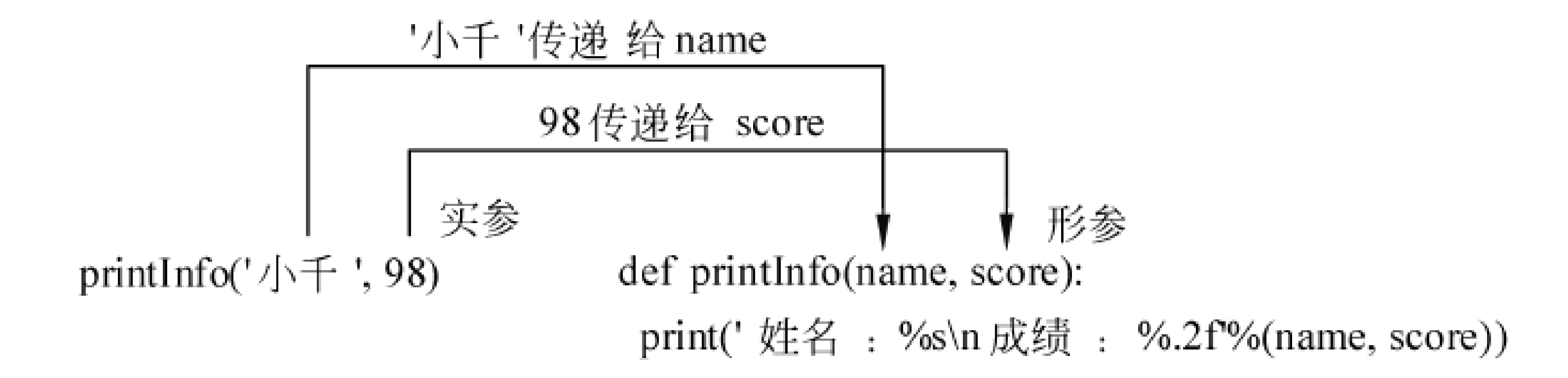


图 7.4 函数参数传递

在图 7.4 中，当函数调用时，实参的传递顺序与定义函数形参的顺序需保持一致。由于实参的顺序与函数定义时形参的位置有关，因此称为位置参数。

7.3.2 关键参数

关键参数指通过对形参赋值传递的参数。关键参数允许函数调用时允许传递实参的顺序与定义函数的形参顺序不一致，因为 Python 解释器能够用形参名匹配实参值，使用户不必记住位置参数的顺序，如例 7-3 所示。

例 7-3 关键参数的使用。

```
1  def printInfo(name, score):
2      print('姓名：%s\n 成绩：%.2f'%(name, score))
3  printInfo('小千', 98)
4  printInfo(score = 98, name = '小千')
```

运行结果如图 7.5 所示。



图 7.5 例 7-3 运行结果

在例 7-3 中，第 1～2 行定义 printInfo()函数。第 4 行调用函数，其参数是根据函数定义时形参的名称进行数据传递的，因此称为关键参数。

7.3.3 默认参数

如果在函数定义时参数列表中的某个形参有值，则称这个参数为默认参数。注意默认参数必须放在非默认参数的右侧，否则函数将出错，如例 7-4 所示。

例 7-4 默认参数的使用。

```
1  def printInfo(name, school = '千锋教育'):
2      print('姓名：%s\t 学校：%s'%(name, school))
3  printInfo('小千')
4  printInfo('小锋', '扣丁学堂')
5  printInfo(school = '好程序员特训营', name = '小明')
```

运行结果如图 7.6 所示。

在例 7-4 中，第 3 行调用函数时，由于定义函数时形参 school 有默认值'千锋教育'，因此调用时可以省略不写该参数。如果想修改默认值，则在调用时传入该参数即可，如本例中的第 4 行。

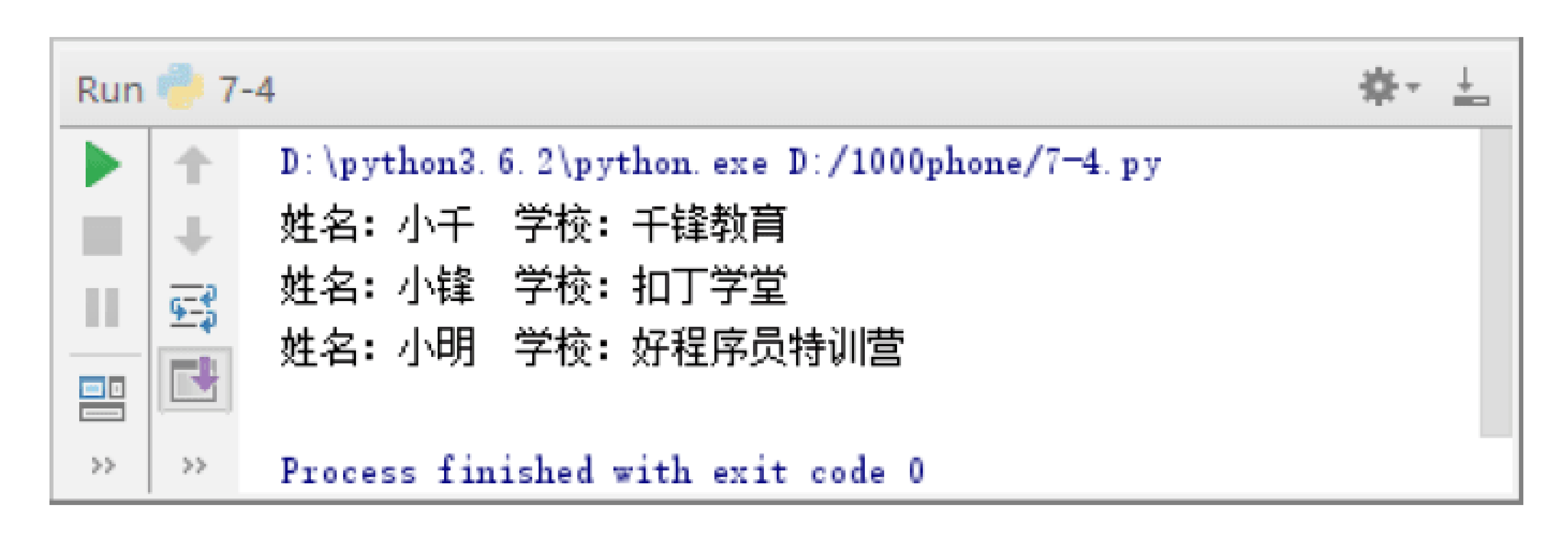


图 7.6　例 7-4 运行结果

默认参数可以让函数的调用更加简单，就如同安装 PC 端软件时，程序会提示用户默认安装路径，当然用户也可以自定义安装路径。

此外，如果将例题中的 name 与 school 调换位置，具体示例如下：

```
def printInfo(school = '千锋教育', name):   # 错误写法
    print('姓名：%s\t 学校：%s'%(name, school))
```

程序运行后，将会报错，如图 7.7 所示。

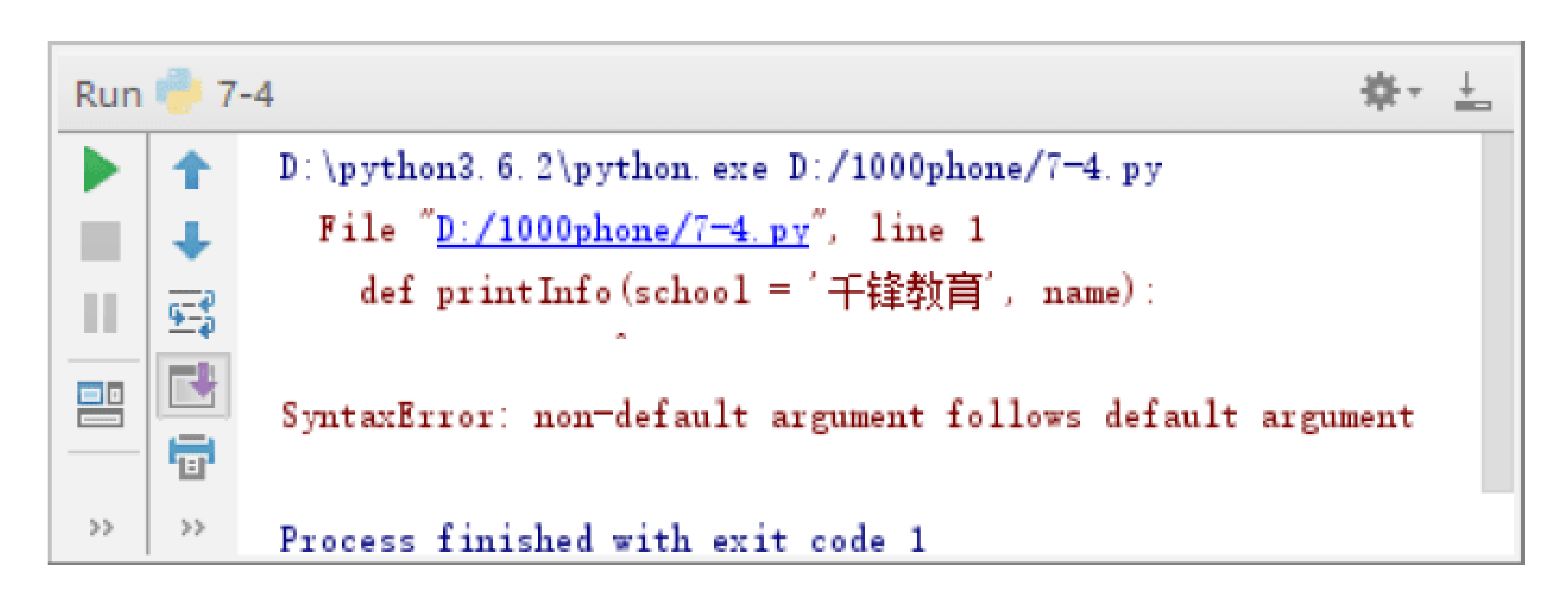


图 7.7　例 7-4 运行结果

7.3.4　不定长参数

在前面对函数的介绍中，一个形参只能接收一个实参。除此之外，函数形参可以接收不定个数的实参，即用户可以给函数提供可变长度的参数，这可以通过在形参前面使用*来实现，如例 7-5 所示。

例 7-5　不定长参数的使用。

```
def mySum(a = 0, b = 0, *args):
    print(a, b, args)
    sum = a + b
    for n in args:
        sum += n
    return sum
print(mySum(1, 2))
print(mySum(1, 2, 3))
print(mySum(1, 2, 3, 4))
```

运行结果如图 7.8 所示。

```
Run 7-5
D:\python3.6.2\python.exe D:/1000phone/7-5.py
1 2 ()
3
1 2 (3,)
6
1 2 (3, 4)
10

Process finished with exit code 0
```

图 7.8　例 7-5 运行结果

在例 7-5 中，第 1 行中加了星号的变量 args 会存放所有未命名的变量参数，其数据类型为元组。第 7 行调用函数时传入 2 个实参，分别对应形参 a 与 b，此时 args 是一个空元组。第 8 行调用函数时传入 3 个参数，此时将第 3 个参数添加到元组中。第 9 行调用函数时传入 4 个参数，此时将后两个参数添加到元组中。

此外，不定长参数还可以接受关键参数并将其存放到字典中，这时需要使用**来实现，如例 7-6 所示。

例 7-6　不定长参数接受关键参数。

```
def mySum(a = 0, b = 0, *args1, **args2):
    print(a, b, args1, args2)
    sum = a + b
    for n in args1:           # 遍历元组
       sum += n
    for key in args2:         # 遍历字典
        sum += args2[key]
    return sum
print(mySum(1, 2, 3, 4))
print(mySum(1, 2, c = 3, d = 4))
print(mySum(1, 2, 3, 4, c = 5, d = 6))
```

运行结果如图 7.9 所示。

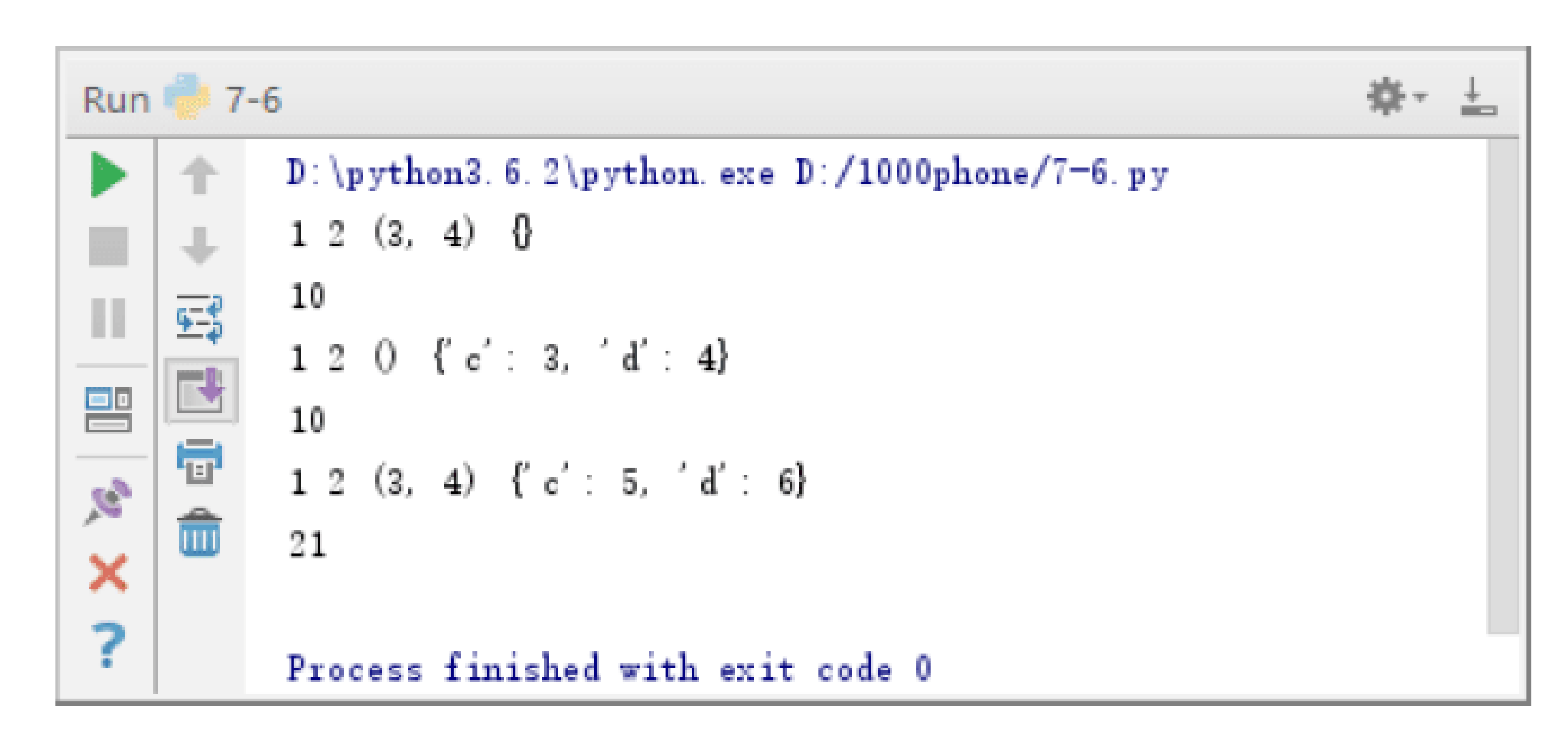


图 7.9　例 7-6 运行结果

在例 7-6 中，第 1 行中加了两个星号的变量 args2 会存放关键参数，其数据类型为字典。第 9 行调用函数时传入 4 个实参，第 3 个参数与第 4 个参数添加到元组 args1 中，此时 args2 是一个空字典。第 10 行调用函数时传入 4 个参数，第 3 个参数与第 4 个参数添加到字典 args2 中，此时 args1 是一个空元组。第 11 行调用函数时传入 6 个参数，第 3 个参数与第 4 个参数添加到元组 args1 中，第 5 个参数与第 6 个参数添加到字典 args2 中。

此外，通过*还可以进行相反的操作，如例 7-7 所示。

例 7-7 *的使用。

```
1   def mySum(a, b, c):
2       return a + b + c
3   tuple1 = (1, 2, 3)
4   print(mySum(*tuple1))
5   list1 = [1, 2, 3]
6   print(mySum(*list1))
```

运行结果如图 7.10 所示。

```
Run 7-7
D:\python3.6.2\python.exe D:/1000phone/7-7.py
6
6

Process finished with exit code 0
```

图 7.10 例 7-7 运行结果

在例 7-7 中，第 4 行在调用函数时在元组 tuple1 前加上星号，此时将 tuple1 中的 3 个元素分别传递给形参 a、b、c。此外，还可以在列表前加星号，如第 6 行。

另外，通过**可以将字典转换为关键参数，如例 7-8 所示。

例 7-8 **的使用。

```
def mySum(a, b, c):
    return a + b + c
dict1 = {'a':1, 'b':2, 'c':3}
print(mySum(**dict1))
```

运行结果如图 7.11 所示。

```
Run 7-8
D:\python3.6.2\python.exe D:/1000phone/7-8.py
6

Process finished with exit code 0
```

图 7.11 例 7-8 运行结果

在例 7-8 中，第 4 行在调用函数时在字典 dict1 前加上两个星号，此时将 dict1 中的 3 个键值对分别转换为关键参数。

此外，需注意上述两种方式的传递顺序，如例 7-9 所示。

例 7-9 *与**的使用。

```
1  def mySum(a, b, c):
2      return a + b + c
3  print(mySum(*(1,),**{'b':2, 'c':3}))
4  # print(mySum(**{'b':2, 'c':3}, *(1,)))  错误写法
```

运行结果如图 7.12 所示。

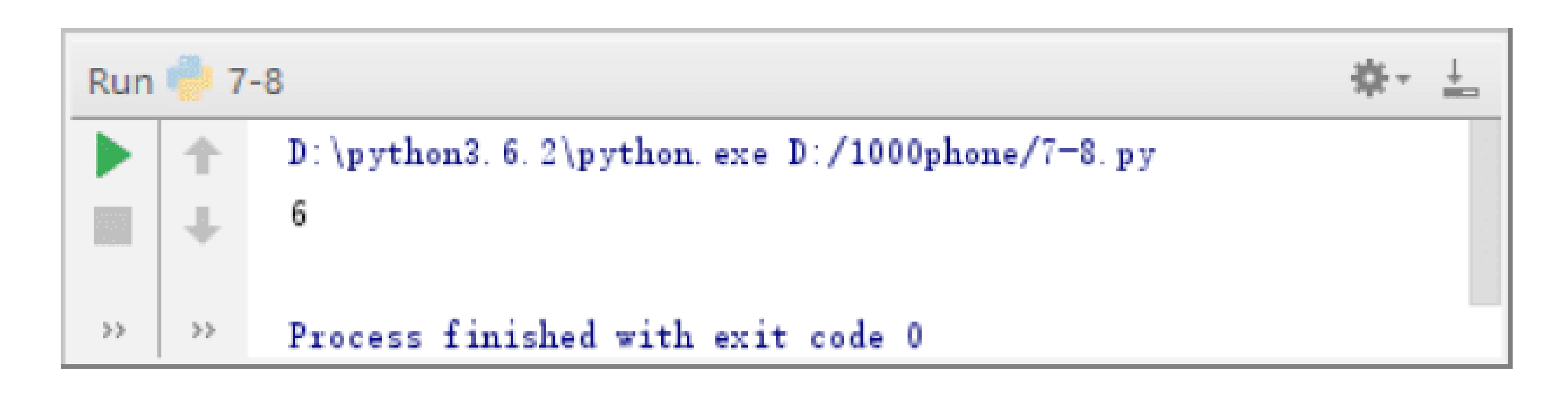


图 7.12 例 7-9 运行结果

在例 7-9 中，第 3 行调用 mySum()函数时，第一个参数在元组前加*，第二个参数在字典前加**，此时形参中 a、b、c 的值分别为 1、2、3。第 4 行交换参数的位置，则会发生错误，因此将此行注释掉。

7.3.5 传递不可变与可变对象

在 Python 中，数字、字符串与元组是不可变类型，而列表、字典是可变类型，两者区别如下：

- 不可变类型——该类型的对象所代表的值不能被改变。当改变某个变量时，由于其所指的值不能被改变，相当于把原来的值复制一份后再改变，这会开辟一个新的地址，变量再指向这个新的地址。
- 可变类型——该类型的对象所代表的值可以被改变。变量改变后，实际上是其所指的值直接发生改变，并没有发生复制行为，也没有开辟出新的地址。

接下来演示调用函数时传递可变与不可变对象，如例 7-10 所示。

例 7-10 调用函数时传递可变与不可变对象。

```
def test1(alist):
    alist.append(5)
    print(alist)
def test2(astr):
    astr += '.com'
    print(astr)
list1 = [1, 2, 3, 4]
str1 = 'codingke'
```

```
9   test1(list1)  # 可变对象
10  test2(str1)   # 不可变对象
11  print(list1, str1)
```

运行结果如图 7.13 所示。

```
Run 7-10
D:\python3.6.2\python.exe D:/1000phone/7-10.py
[1, 2, 3, 4, 5]
codingke.com
[1, 2, 3, 4, 5] codingke

Process finished with exit code 0
```

图 7.13 例 7-10 运行结果

在例 7-10 中，第 9 行调用 test1()函数，实参为可变对象，第 10 行调用 test2()函数，实参为不可变对象。从程序运行结果可发现，可变对象可以被修改，但不可变对象不能被修改。

7.4 函数的返回值

函数调用时的参数传递实现了从函数外部向函数内部输入数据，而函数的 return 语句实现了从函数内部向函数外部输出数据。

此处需注意，如果函数定义时省略 return 语句或者只有 return 而没有返回值，则 Python 将认为该函数以“return None”结束，None 代表没有值，如例 7-11 所示。

例 7-11 函数定义时省略 return 语句。

```
1   def output():
2       print('拼搏到无能为力，坚持到感动自己!')
3   print(output())
```

运行结果如图 7.14 所示。

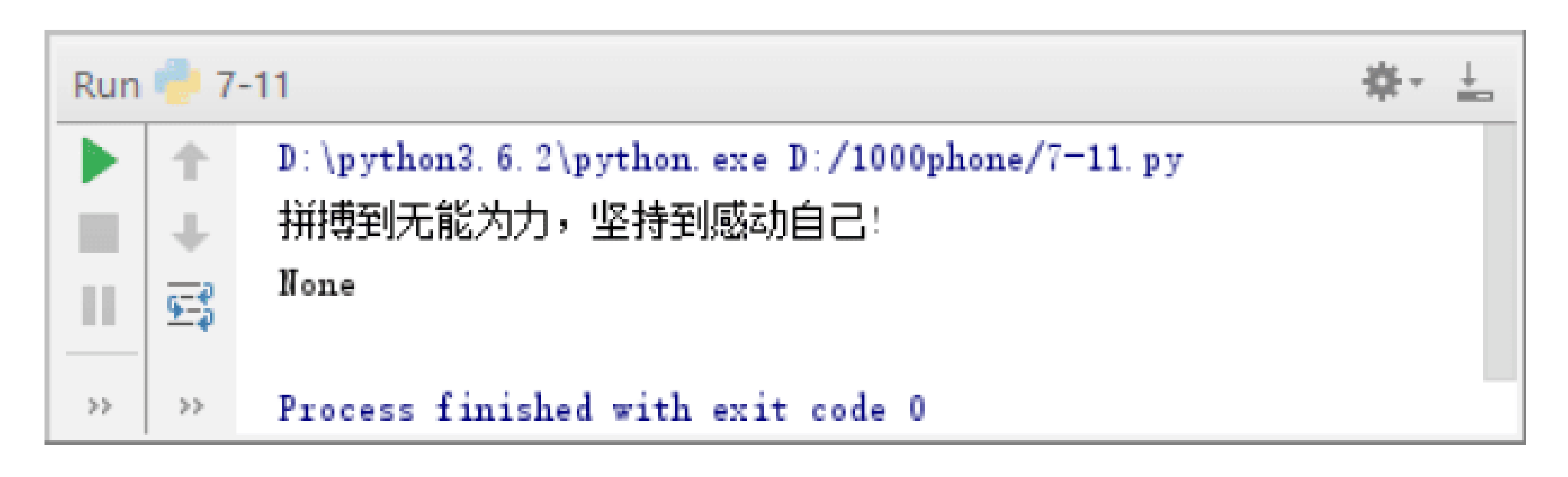


图 7.14 例 7-11 运行结果

在例 7-11 中，第 3 行通过 print()函数打印 output()函数的返回值，此时输出 None。

return 语句可以放置在函数中任何位置，当执行到第一个 return 语句时，程序返回到调用程序处接着执行，此时不会执行该函数中 return 语句后的代码，如例 7-12 所示。

例 7-12 return 语句的用法。

```
1  def myMax(a, b):
2      if a > b:
3          return a
4      else:
5          return b
6      print(a, b)
7  print(myMax(2, 5))
```

运行结果如图 7.15 所示。

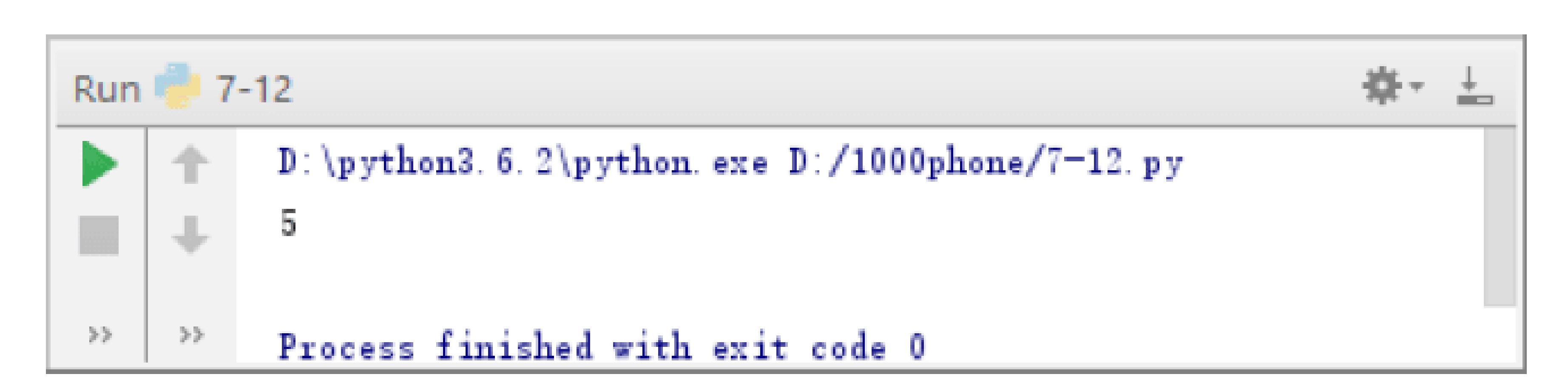


图 7.15　例 7-12 运行结果

在例 7-12 中，第 7 行调用函数时，将实参 2、5 分别传递给形参 a、b，程序跳转到第 1 行处执行，由于 a 小于 b，因此执行 else 后的 return 语句，此时函数调用结束，不会执行第 6 行语句，最终输出函数的返回值为 5。

当函数具有多个返回值时，如果只用一个变量来接收返回值，函数返回的多个值实际上构成了一个元组，如例 7-13 所示。

例 7-13 函数返回多个值。

```
1  def calculate(a, b):
2      return a + b, a - b, a * b, a / b
3  x = calculate(6, 3)
4  print(x)
5  a1, b1, c1, d1 = calculate(6, 3)
6  print(a1, b1, c1, d1)
```

运行结果如图 7.16 所示。

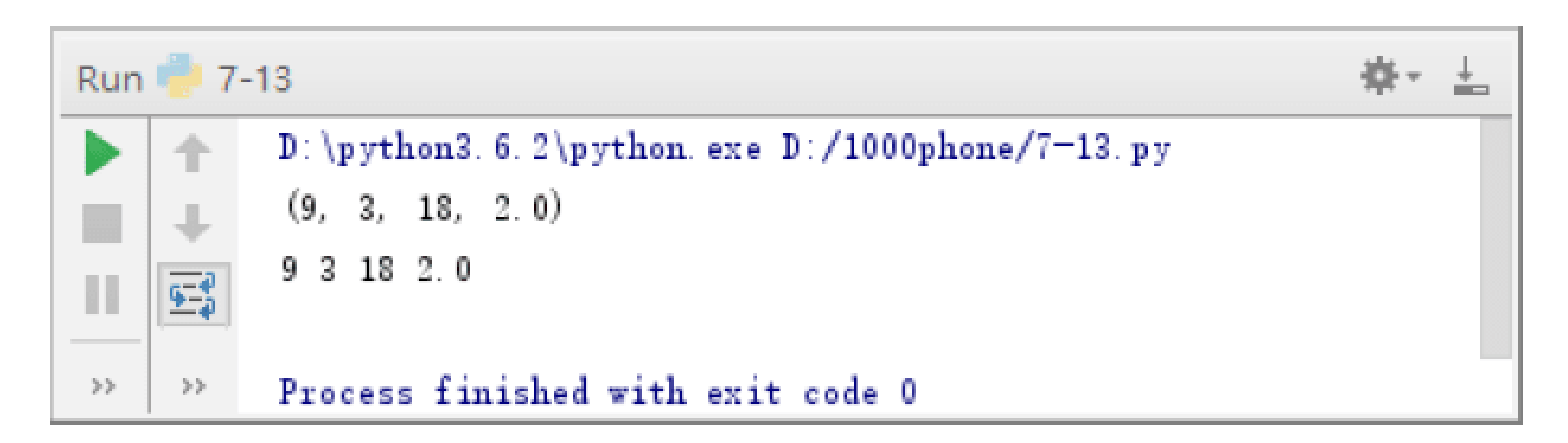


图 7.16　例 7-13 运行结果

在例 7-13 中，第 2 行函数通过 return 语句返回 4 个值，第 3 行通过一个变量接收函数 calculate()的返回值，第 4 行打印该变量，输出一个元组。第 5 行利用多变量同时赋值语句来接收多个返回值。

7.5 变量的作用域

变量起作用的代码范围称为变量的作用域，与变量定义的位置密切相关，按照作用域的不同，变量可分为局部变量和全局变量。

7.5.1 局部变量

在函数内部定义的普通变量只在函数内部起作用，称为局部变量。当函数执行结束后，局部变量自动删除，不可以再使用，如例 7-14 所示。

例 7-14 局部变量。

```
1   def fun1():
2       x = 1 # 局部变量
3       print('fun1()函数中的 x 为%d'%x)
4   def fun2():
5       x = 2 # 局部变量
6       print('fun2()函数中的 x 为%d'%x)
7   fun1()
8   fun2()
9   # print(x)
```

运行结果如图 7.17 所示。

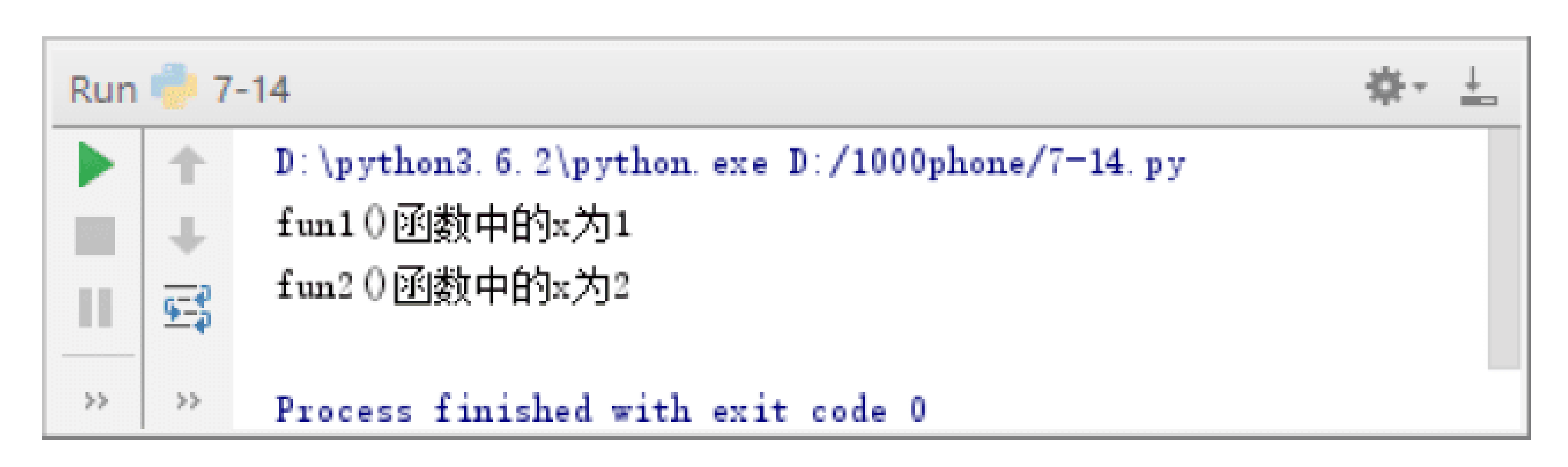


图 7.17 例 7-14 运行结果

在例 7-14 中，第 2 行与第 5 行定义的 x 虽然同名，但属于不同的作用域，两者互不影响。第 9 行在函数外访问局部变量，程序运行时会报 x 未定义的错误，因此将此行注释掉。

7.5.2 全局变量

如果需要在函数内部给一个定义在函数外的变量赋值，那么这个变量的作用域不能是局部的，而应该是全局的。能够同时作用于函数内外的变量称为全局变量，它通过 global 关键字来声明，如例 7-15 所示。

例 7-15 全局变量。

```
x = 2 # 全局变量
```

```
def fun():
    global x # 使用 global 关键字声明
    x += 1
    print('x = %d'%x)
fun()
print(x)
```

运行结果如图 7.18 所示。

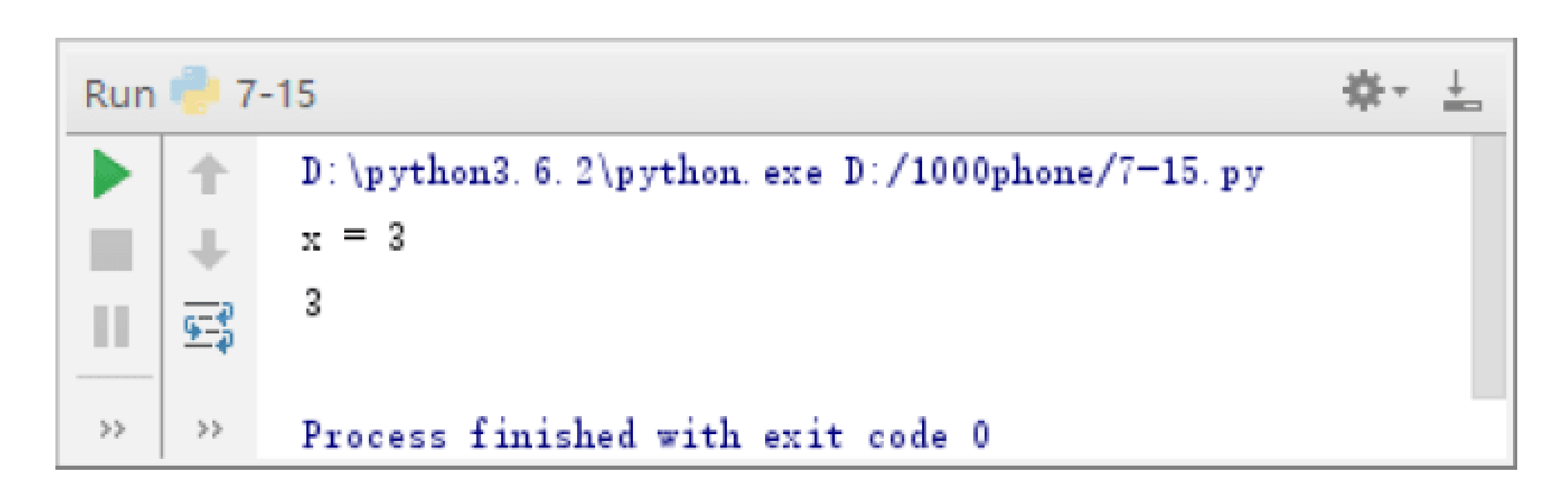


图 7.18 例 7-15 运行结果

在例 7-15 中，变量 x 已在函数外定义，在函数 fun()内修改外部变量 x，则必须在函数内用 global 关键字将该变量声明为全局变量。

此处需注意，如果不使用 global 声明，则在函数中访问的是局部变量，如例 7-16 所示。

例 7-16 函数中的局部变量。

```
1 x = 2 # 全局变量
2 def fun():
3     x = 3 # 局部变量
4     print('x = %d'%x)
5 fun()
6 print(x)
```

运行结果如图 7.19 所示。

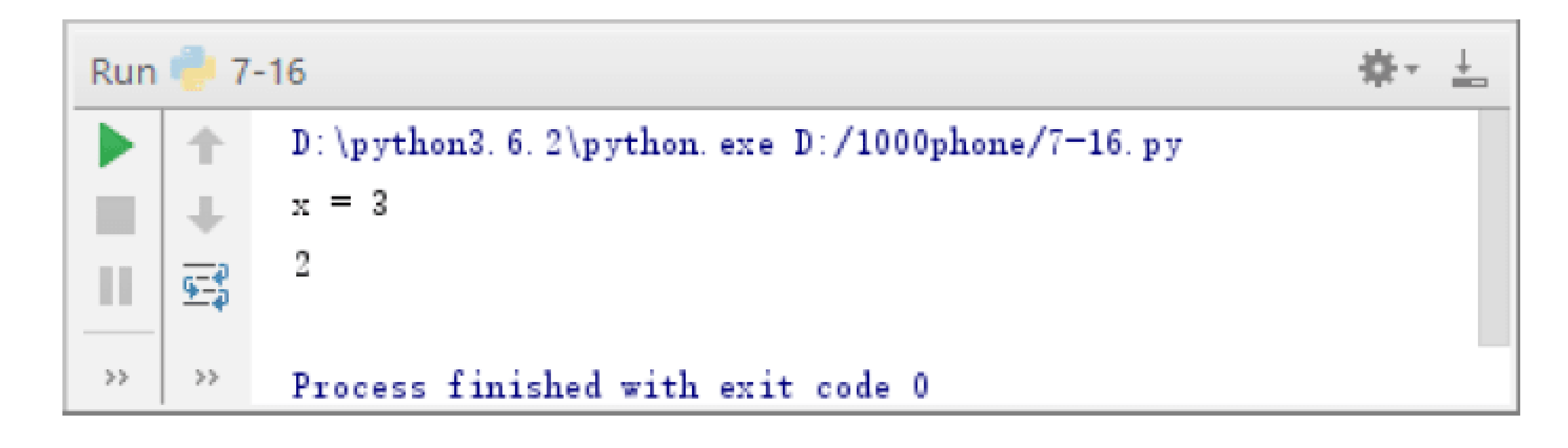


图 7.19 例 7-16 运行结果

在例 7-16 中，第 1 行 x 为全局变量，第 3 行 x 为局部变量，只在 fun()函数内有效。第 4 行打印局部变量 x 的值 3，第 6 行打印全局变量 x 的值 2。

此外，使用内置函数 globals()与 locals()可以查看局部变量与全局变量，如例 7-17 所示。

例 7-17 查看局部变量与全局变量。

```
x = 1            # 全局变量
def fun():
    x, y = 2, 3  # 局部变量
    print(x, y)
    global z     # 全局变量
    z = 1
    print(locals())
fun()
print(x)
print(globals())
```

运行结果如图 7.20 所示。

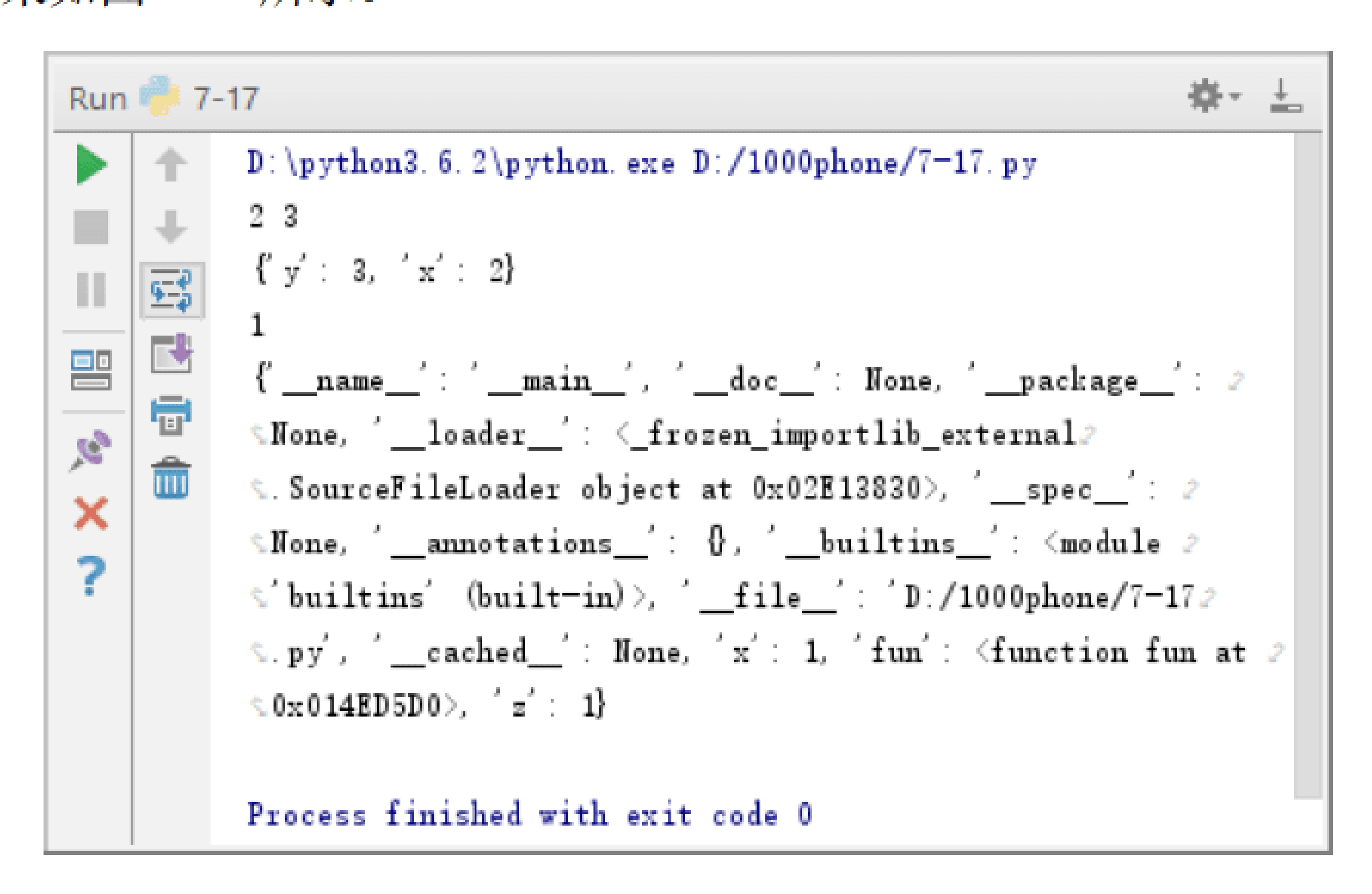


图 7.20 例 7-17 运行结果

在例 7-17 中，函数 globals()与 locals()分别返回一个字典，通过打印字典中的元素，可以查看全局变量与局部变量。另外，在函数内部可以通过 global 关键字直接将一个变量声明为全局变量，即使在函数外没有定义，则该函数执行后，这个变量将成为全局变量，如本例中的变量 z。

7.6 函数的嵌套调用

Python 语言允许在函数定义中出现函数调用，从而形成函数的嵌套调用，如例 7-18 所示。

例 7-18 函数的嵌套调用。

```
def fun1():
    print('fun1()函数开始')
```

```
3        print('fun1()函数结束')
4    def fun2():
5        print('fun2()函数开始')
6        fun1()
7        print('fun2()函数结束')
8    fun2()
```

运行结果如图 7.21 所示。

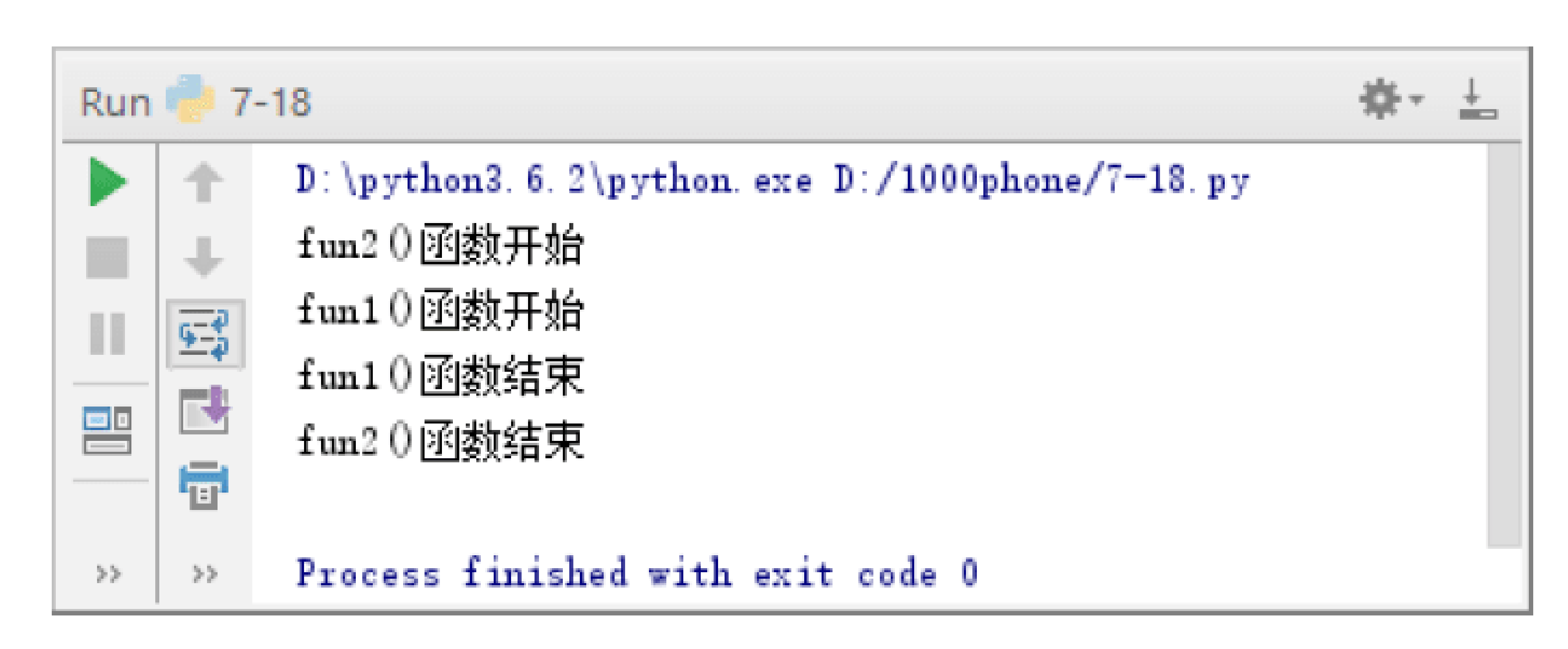


图 7.21 例 7-18 运行结果

在例 7-18 中，第 6 行在 fun2()函数中调用 fun1()函数，程序执行时会跳转到 fun1()函数处去执行，执行完 fun1()后，接着执行 fun2()函数中剩余的代码，如图 7.22 所示。

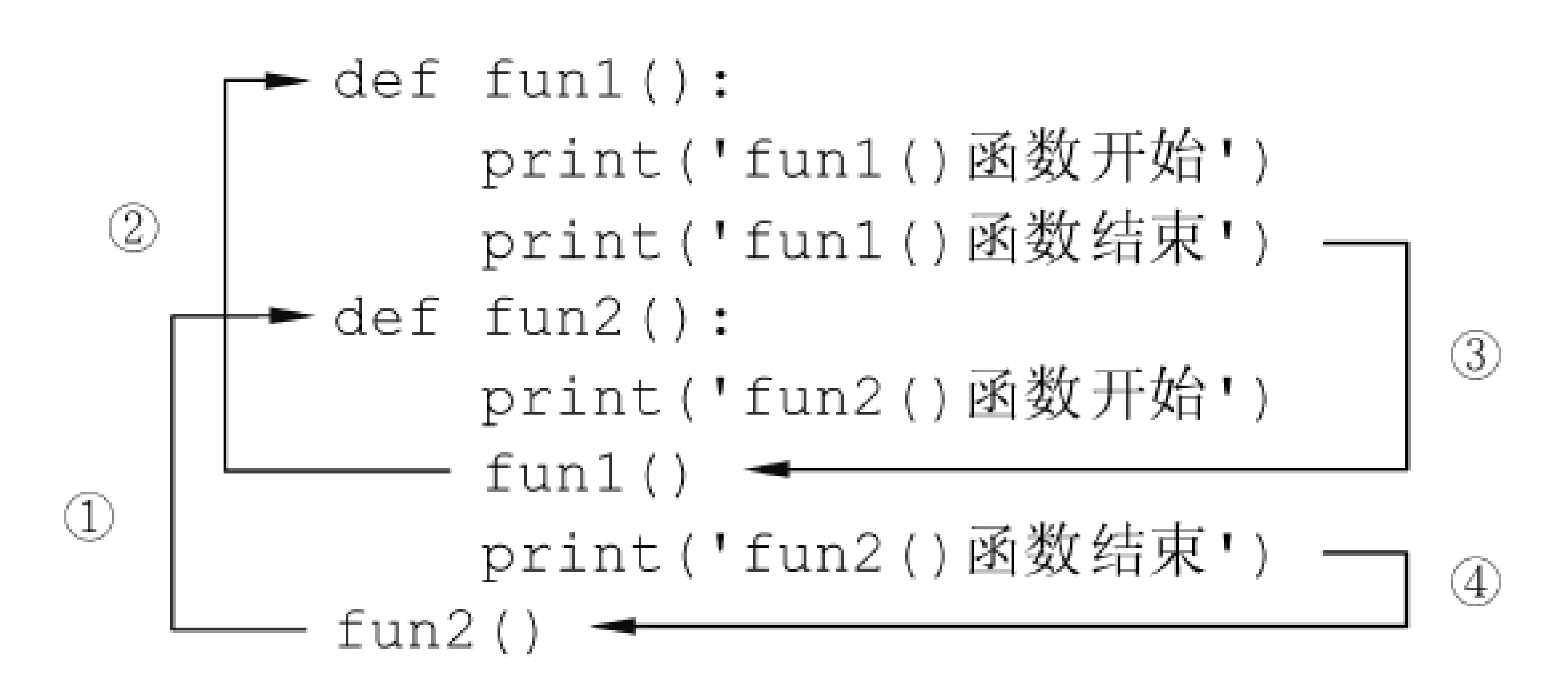


图 7.22 函数的嵌套调用执行过程

7.7 函数的递归调用

在函数的嵌套调用中，一个函数除了可以调用其他函数外，还可以调用自身，这就是函数的递归调用。递归必须要有结束条件，否则会无限地递归（Python 默认支持 997 次递归，多于这个次数将终止）。

接下来演示函数的递归调用，如例 7-19 所示。

例 7-19 函数的递归调用。

```
def f(n):
    '''
    计算阶乘公式:
```

```
4            0! = 1
5            n! = n * (n -1)!, n > 0
6        转化为递归函数：
7            f(0) = 1
8            f(n) = n * f(n - 1), n > 0
9        '''
10       if n == 0:
11           return 1
12       return n * f(n - 1)
13   print('4! = %d'%f(4))
```

运行结果如图 7.23 所示。

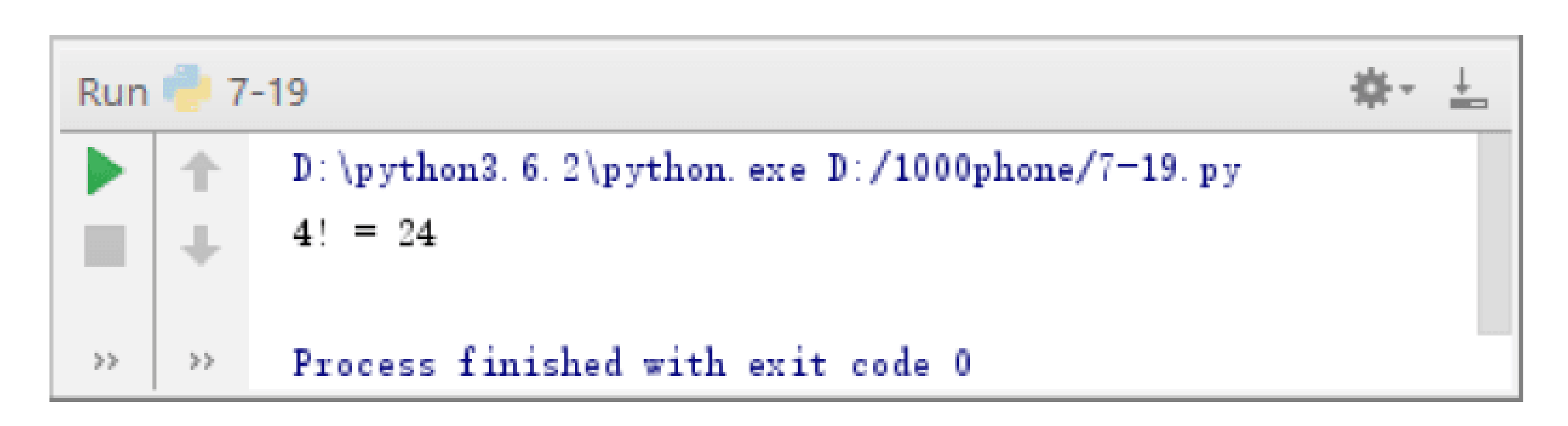


图 7.23　例 7-19 运行结果

在例 7-19 中，第 10 行到第 12 行定义 f()函数用于计算阶乘。当 n == 0 时，程序立即返回结果，这种简单情况称为结束条件。如果没有结束条件，就会出现无限递归。当 n > 0 时，就将这个原始问题分解成计算 n–1 阶乘的子问题，持续分解，直到问题达到结束条件为止，就将结果返回给调用者，然后调用者进行计算并将结果返回给它自己的调用者，该过程持续进行，直到结果返回原始调用者为止。原始问题就可以通过将 f(n–1)的结果乘以 n 得到，这种调用过程就称为递归调用，如图 7.24 所示。

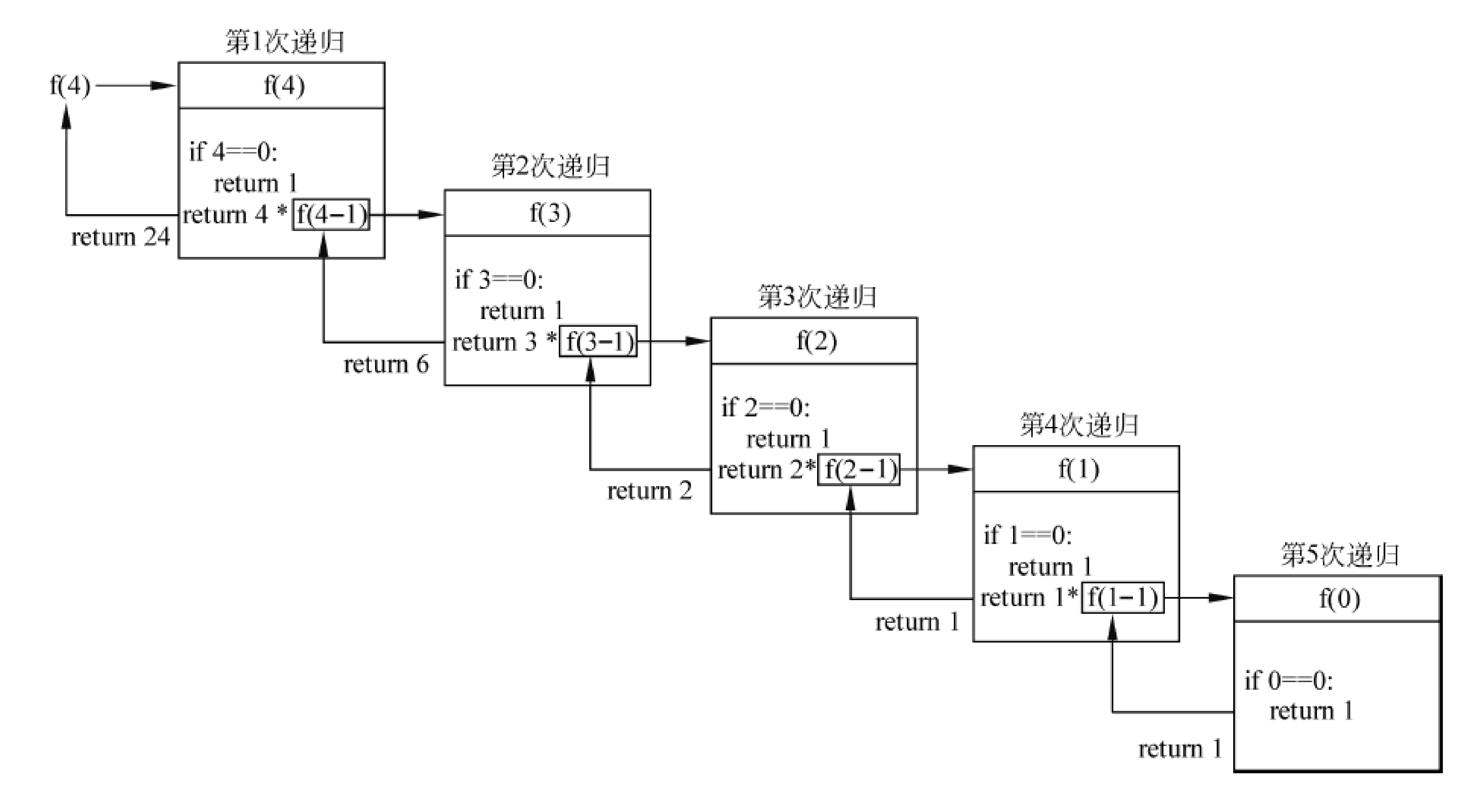


图 7.24　函数的递归调用

7.8 小 案 例

7.8.1 案例一

编写两个函数，一个函数接收一个整数 num 为参数，生成杨辉三角形的前 num 行数据，另一个函数接收生成的杨辉三角形并按如图 7.25 所示的形式输出。

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
```

图 7.25 杨辉三角形

在图 7.25 中，列出了杨辉三角形的前 9 行。每一层左右两端的数都是 1 并且左右对称，从第 1 层开始，每个不位于左右两端的数等于上一层左右两个数相加之和，具体实现如例 7-20 所示。

例 7-20 输出杨辉三角形的前 num 行数据。

```
# 生成杨辉三角形
def triangle(num):
    triangle=[[1]]
    for i in range(2, num + 1):
        triangle.append([1]*i)
        for j in range(1, i - 1):
            triangle[i-1][j] = triangle[i-2][j] + triangle[i-2][j-1]
    return triangle
# 打印杨辉三角形
def printtriangle(triangle):
    width = len(str(triangle[-1][len(triangle[-1]) // 2])) + 3
    column = len(triangle[-1]) * width
    for sublist in triangle:
        result = []
        for contents in sublist:
            # 控制间距
            result.append('{0:^{1}}'.format(str(contents), width))
        # 控制缩进
```

```
        print('{0:^{1}}'.format(''.join(result), column))
num = int(input('请输入行数:'))
triangle = triangle(num)
printtriangle(triangle)
```

运行结果如图 7.26 所示。

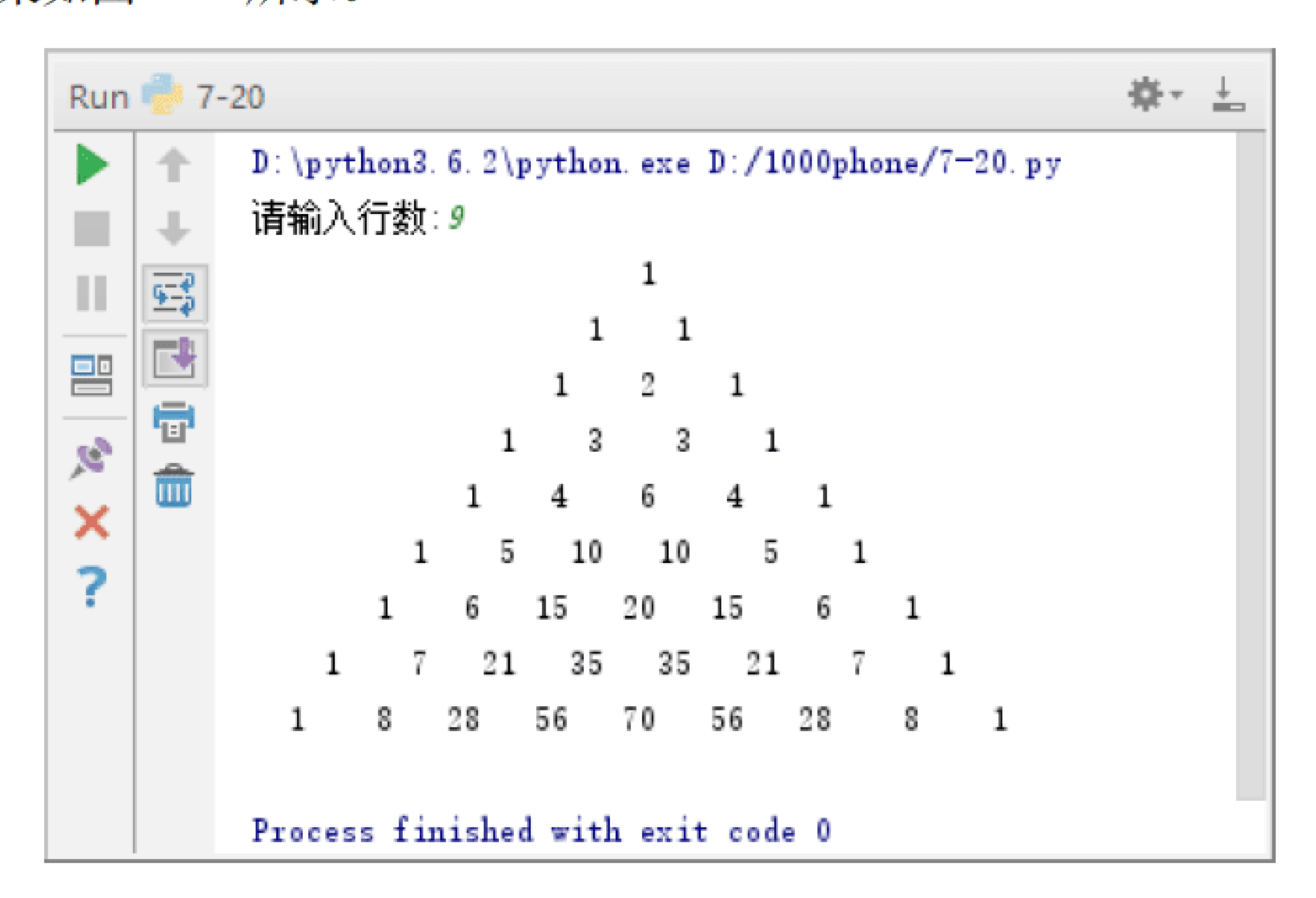


图 7.26　例 7-20 运行结果

在例 7-20 中，triangle()函数中使用列表来存储杨辉三角形中的数据，列表中的每个元素又是一个列表，其中存储杨辉三角形的某一行数据。printtriangle()函数中的 format()函数用于格式化字符串，在此处只需简单了解即可。

7.8.2　案例二

汉诺塔问题是源于印度一个古老传说，大梵天创造世界时，在世界中心贝拿勒斯的圣庙中做了 3 根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘（称为汉诺塔）。大梵天命令婆罗门把圆盘从一根柱子上按大小顺序重新摆放在另一根柱子上，并规定在 3 根柱子之间一次只能移动一个圆盘且小圆盘上不能放置大圆盘，如图 7.27 所示。

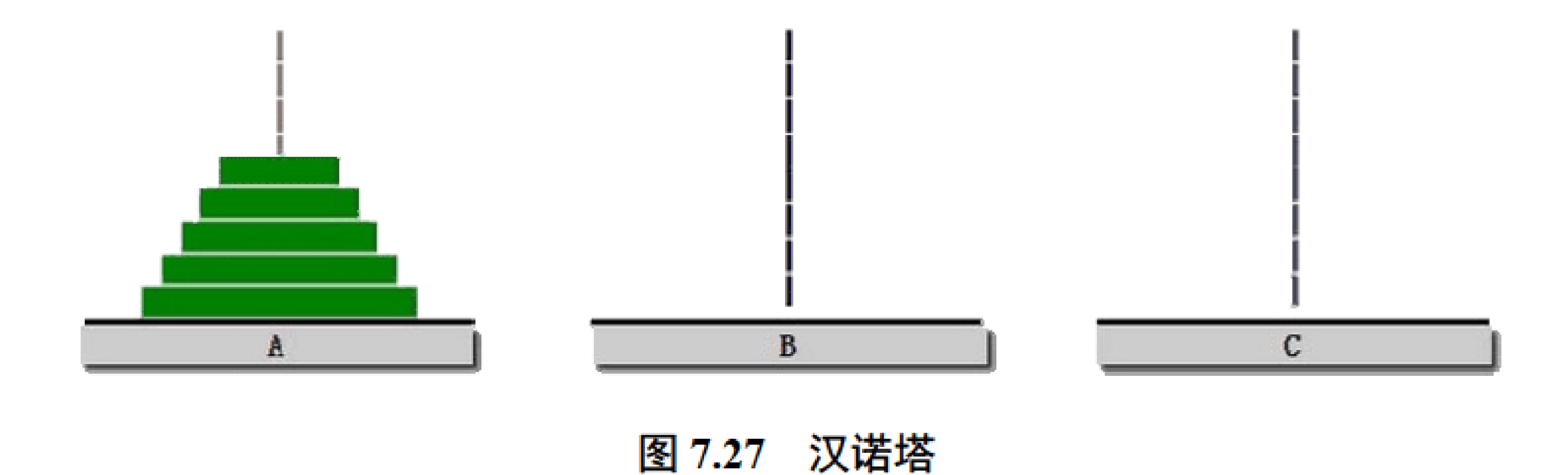


图 7.27　汉诺塔

假设使用 1，2，…，n 标记 n 个大小互不相同的圆盘，A、B、C 标记 3 个柱子，初

始状态时所有圆盘都放在 A 柱子上，最终状态时所有的圆盘都放在 C 柱子上，柱子 B 作为中间缓冲。

当 n = 1 时，即只有 1 个圆盘，此时可以直接把这个圆盘从柱子 A 移动到柱子 C，这也是递归的终止条件。

当 n > 1 时，依次解决以下 3 个子问题：

- 借助柱子 C 将前 n−1 个盘子从柱子 A 移到柱子 B；
- 将圆盘 n 从柱子 A 移 到柱子 C；
- 借助柱子 A 将前 n−1 个圆盘从柱子 B 移到柱子 C。

汉诺塔问题的具体实现如例 7-21 所示。

例 7-21　汉诺塔问题。

```
def hanoi(n, a, b, c):
    if n == 1:
        # 当仅有 1 个圆盘,直接将圆盘从柱子 a 移动到柱子 c 上
        print(n, a, '->',c)
    else:
        # 将 n-1 个圆盘从柱子 a 移动到柱子 b 上
        hanoi(n-1, a, c, b)
        # 将最大的圆盘从柱子 a 移动到柱子 c 上
        print(n, a, '->',c)
        # 将 n-1 个圆盘从柱子 b 移动到柱子 c 上
        hanoi(n-1, b, a, c)
n = int(input('请输入圆盘数：'))
hanoi(n, 'A', 'B', 'C')
```

运行结果如图 7.28 所示。

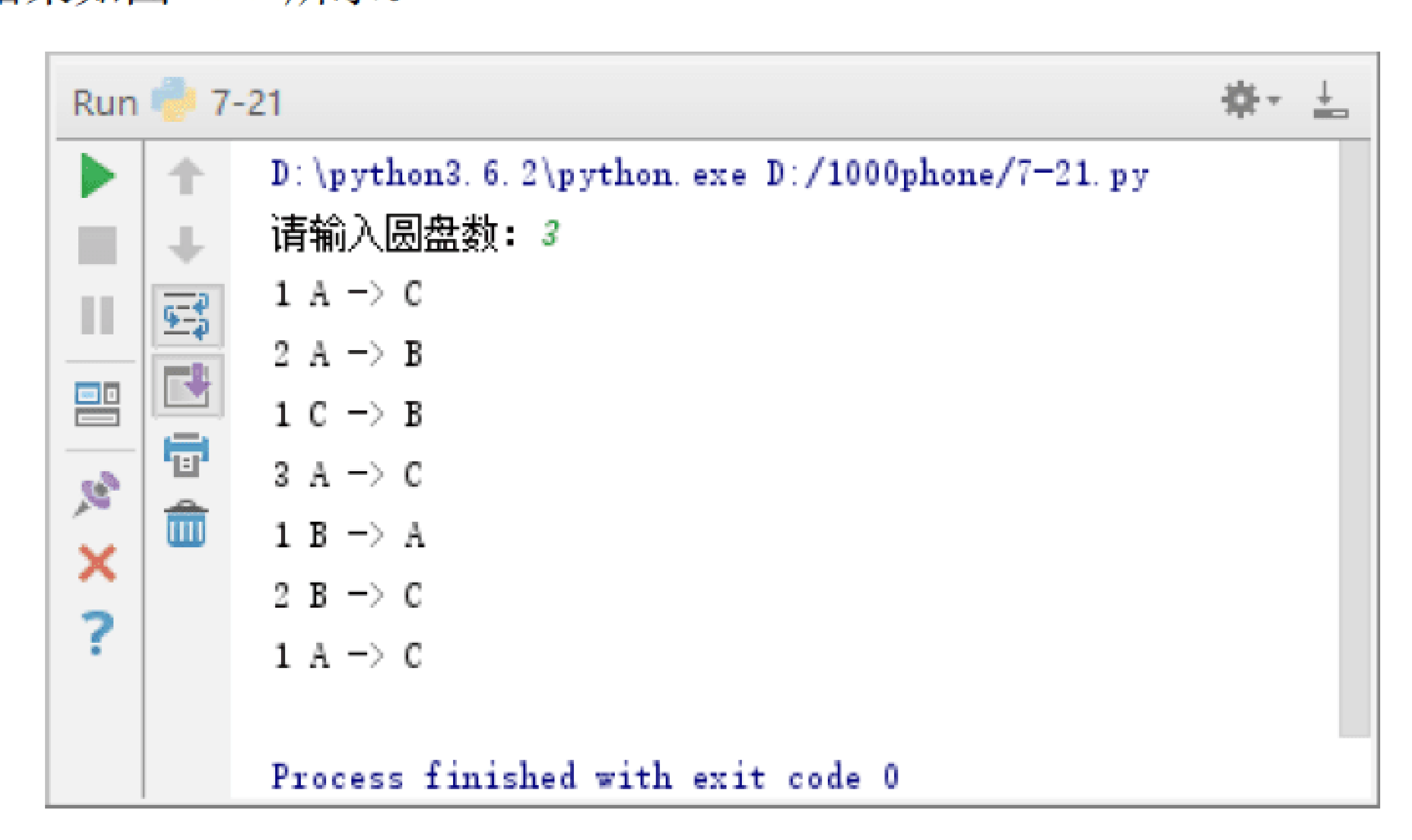


图 7.28　例 7-21 运行结果

在例 7-21 中，使用函数的递归调用解决汉诺塔问题。使用递归需抓住两个关键点：一是递归的结束条件，二是递归的规律。

7.9 本章小结

本章主要介绍了 Python 中函数的基本知识，包括函数的定义、函数的参数、函数的返回值、函数的嵌套调用与递归调用。通过本章的学习，应熟练掌握如何自定义函数、如何设置函数的参数。另外，在实际编写程序时，应尽量使用函数来简化一些代码，提高代码的可读性、可重用性及可维护性。

7.10 习　题

1．填空题

（1）________关键字表示定义一个函数。
（2）通过________语句可以返回函数值并退出函数。
（3）在函数内部定义的变量为________变量。
（4）省略 return 语句的函数将返回________。
（5）在函数内部修改全局变量，需要使用________关键字声明。

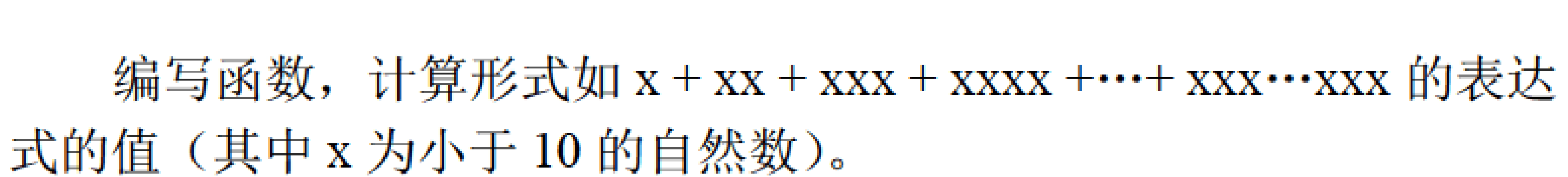
2．选择题

（1）若想查看一个函数的文档字符串，则可以通过（　　）属性实现。
A．__doc__　B．__name__　C．__func__　D．__str__
（2）若只用一个变量来接收函数返回的多个值，则这个变量类型为（　　）。
A．列表　B．元组　C．字典　D．集合
（3）下列属于可变类型的是（　　）。
A．数字　B．字符串　C．列表　D．元组
（4）在函数定义时某个形参有值，则称这个参数为（　　）。
A．不定长参数　B．位置参数　C．关键参数　D．默认参数
（5）（　　）函数可以得到程序中所有的全局变量。
A．globals()　B．locals()　C．global()　D．local()

3．思考题

（1）简述局部变量与全局变量的区别。
（2）简述位置参数与关键参数的区别。

4．编程题


编写函数，计算形式如 x + xx + xxx + xxxx +…+ xxx…xxx 的表达式的值（其中 x 为小于 10 的自然数）。



第 8 章 chapter 8

函数（下）

本章学习目标

- 理解间接调用函数。
- 掌握匿名函数。
- 掌握闭包与装饰器。
- 理解偏函数。
- 掌握常用的内建函数。

第 7 章讲解了函数的基本知识，本章将带领大家继续深入学习函数，只有了解函数高级的用法，才能更好地编写出简洁的代码，同时也便于阅读优秀代码并借鉴到自己程序中。

8.1 间接调用函数

前面调用函数时，使用函数名加参数列表的形式调用。除此之外，还可以将函数名赋值给一个变量，再通过变量名加参数列表的形式间接调用函数，如例 8-1 所示。

例 8-1 变量名加参数列表的形式间接调用函数。

```
def output(message):
    print(message)
output('直接调用 output()函数！')
x = output
x('间接调用 output()函数！')
```

运行结果如图 8.1 所示。

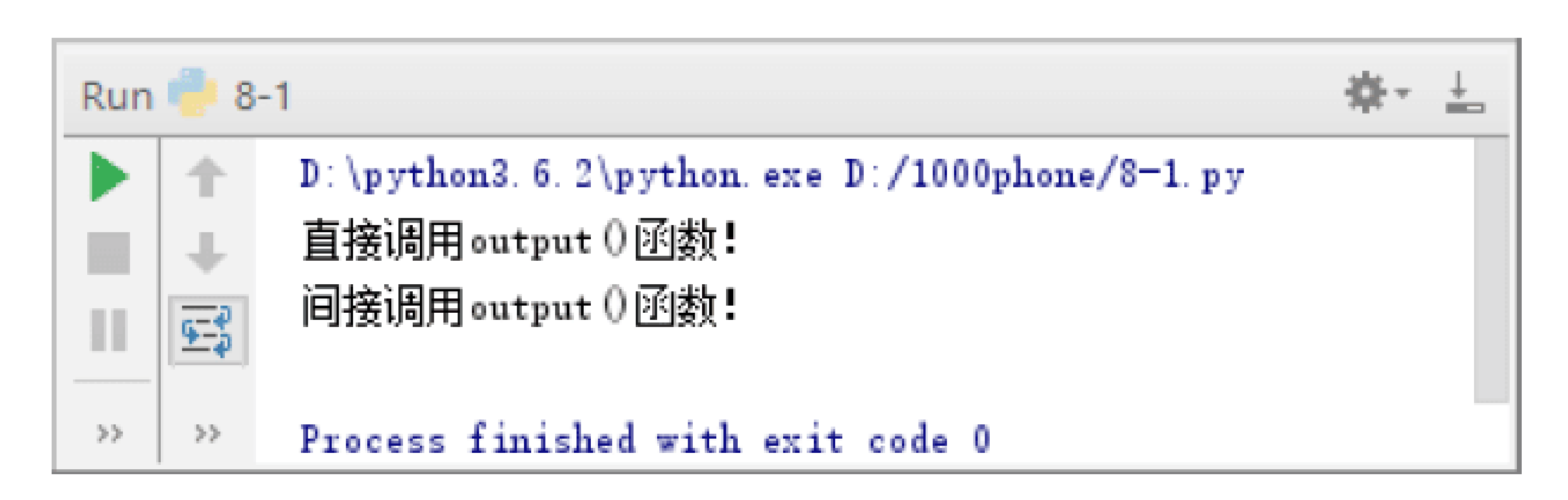


图 8.1 例 8-1 运行结果

在例 8-1 中，第 3 行通过函数名直接调用 output()函数，第 5 行通过变量 x 间接调用 output()函数。

大家可能会疑惑：间接调用函数有何用处?这种用法可以使一个函数作为另一个函数的参数，如例 8-2 所示。

例 8-2　一个函数作为另一个函数的参数。

```
1   def output(message):
2       print(message)
3   def test(func, arg):
4       func(arg)
5   test(output, '一个函数作为另一个函数的参数')
```

运行结果如图 8.2 所示。

```
Run 8-2
D:\python3.6.2\python.exe D:/1000phone/8-2.py
一个函数作为另一个函数的参数

Process finished with exit code 0
```

图 8.2　例 8-2 运行结果

在例 8-2 中，第 5 行将 output()函数的函数名作为参数传入 test()函数中，第 4 行相当于间接调用 output()函数。

另外，函数名还可以作为其他数据类型的元素，如例 8-3 所示。

例 8-3　函数名作为其他数据类型的元素。

```
def output(message):
    print(message)
list1 = [(output, '千锋教育'), (output, '扣丁学堂')]
for (func, arg) in list1:
    func(arg)
```

运行结果如图 8.3 所示。

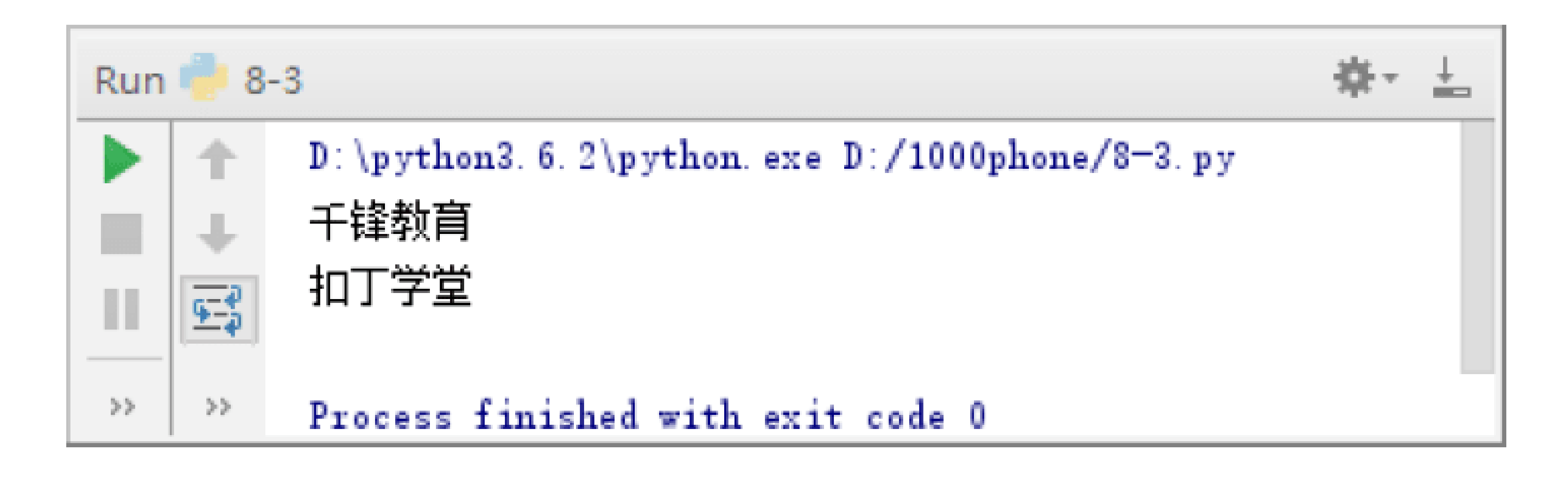


图 8.3　例 8-3 运行结果

在例 8-3 中，第 3 行列表 list1 中的每个元素为元组，元组中的第一个元素为函数 output()的函数名，第 4 行遍历列表 list1，第 5 行相当于间接调用 output()函数。

8.2 匿名函数

匿名函数是指没有函数名称的、临时使用的微函数。它可以通过 lambda 表达式来声明，其语法格式如下：

```
lambda [arg1 [, arg2,…, argn]] : 表达式
```

其中，“[arg1 [, arg2,…, argn]]”表示函数的参数，“表达式”表示函数体。lambda 表达式只可以包含一个表达式，其计算结果可以看作是函数的返回值。虽然 lambda 表达式不允许包含其他复杂的语句，但在表达式中可以调用其他函数。

接下来演示 lambda 表达式的使用，如例 8-4 所示。

例 8-4 lambda 表达式的使用。

```
1   sum = lambda num1, num2 : num1 + num2
2   print(sum(4, 5))
```

运行结果如图 8.4 所示。

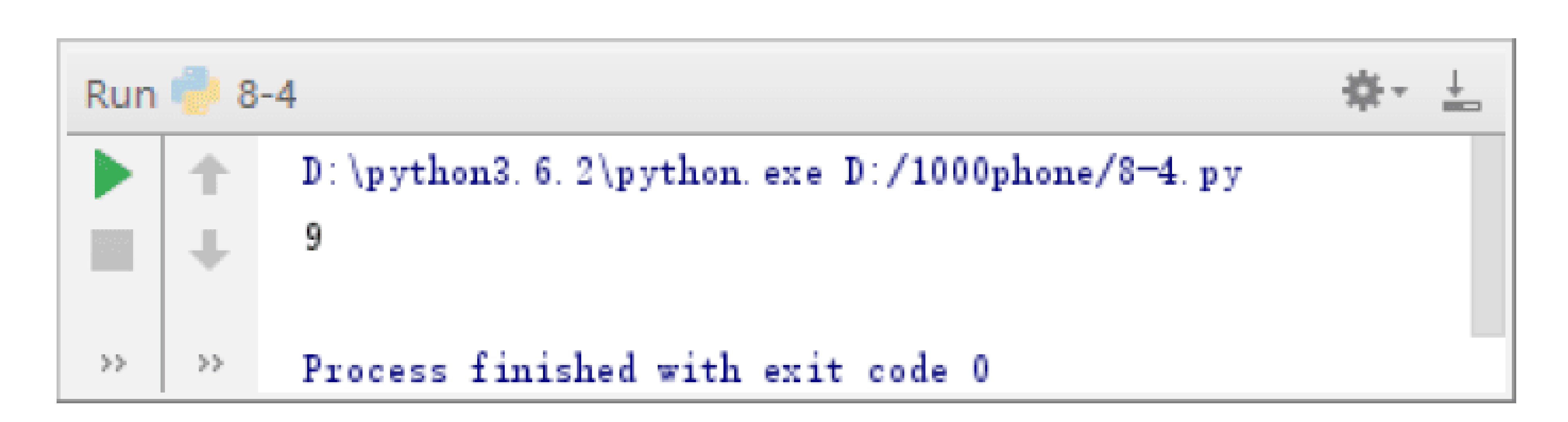


图 8.4 例 8-4 运行结果

在例 8-4 中，第 1 行使用 lambda 表达式声明匿名函数并赋值给 sum，相等于这个函数有了函数名 sum，该行相当于以下代码：

```
def sum(num1, num2):
    return num1 + num2
```

使用 lambda 表达式声明的匿名函数也可以作为自定义函数的实参，如例 8-5 所示。

例 8-5 lambda 表达式声明的匿名函数作为自定义函数的实参。

```
def fun(num1, num2, func):
    return func(num1, num2)
print(fun(8, 6, lambda num1, num2 : num1 + num2))
print(fun(8, 6, lambda num1, num2 : num1 - num2))
```

运行结果如图 8.5 所示。

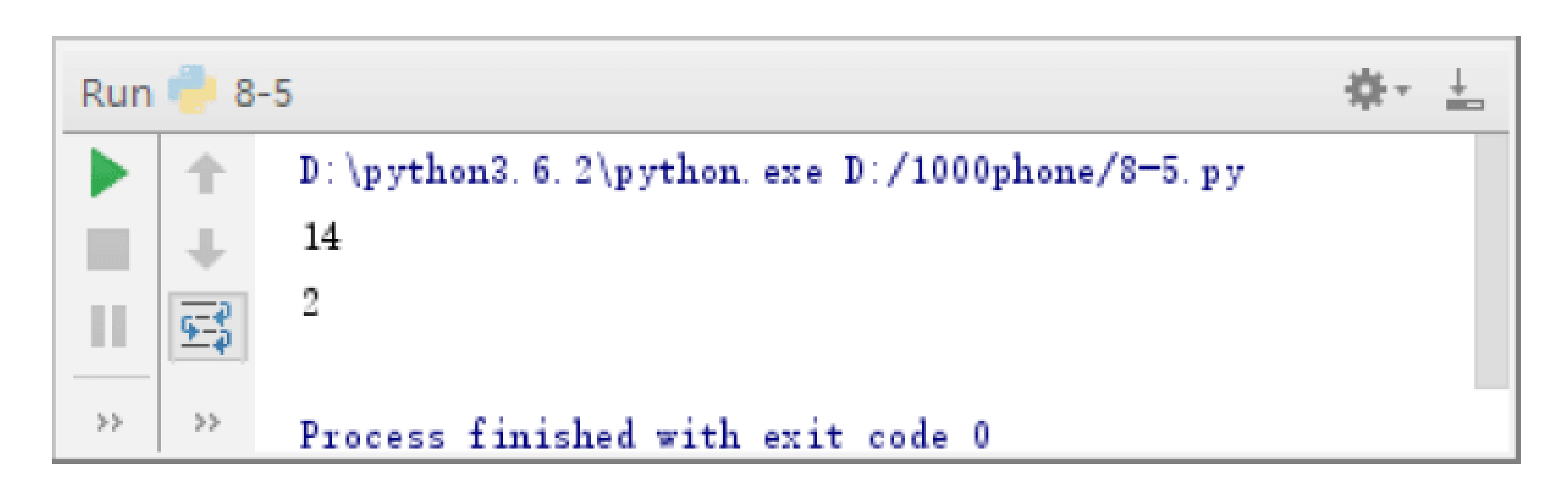


图 8.5　例 8-5 运行结果

在例 8-5 中，第 3 行使用 lambda 表达式作为 fun()函数的实参，第 2 行相当于间接调用 lambda 表达式声明的匿名函数。

此外，lambda 表达式声明的匿名函数还可以作为内建函数的实参，如例 8-6 所示。

例 8-6　lambda 表达式声明的匿名函数作为内建函数的实参。

```
1   info = [
2       {'name':'xiaoqian', 'score':'99'},
3       {'name':'xiaofeng', 'score':'90'},
4       {'name':'xiaoming', 'score':'95'}
5   ]
6   # 按姓名字母由大到小排序
7   info1 = sorted(info, key = lambda x:x['name'], reverse = True)
8   print(info1)
9   # 按分数由小到大排序
10  info2 = sorted(info, key = lambda x:x['score'])
11  print(info2)
```

运行结果如图 8.6 所示。

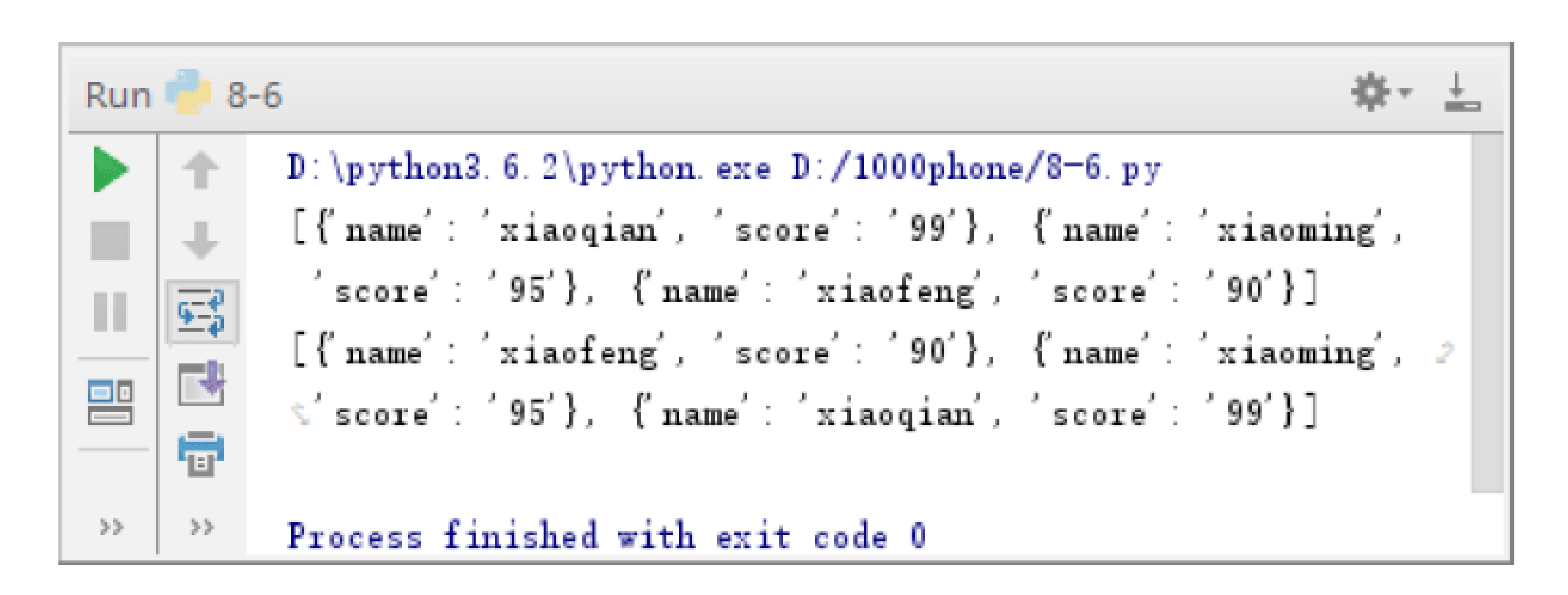


图 8.6　例 8-6 运行结果

在例 8-6 中，第 7 行使用“key = lambda x:x['name']”作为 sorted()函数的关键参数，此时 sorted()函数将列表 info 中的元素按照'name'对应的值进行排序并赋值给 info1，“reverse = True”指定排序规则为从大到小排序。

lambda 表达式表示一个匿名函数，也可以作为列表或字典的元素，如例 8-7 所示。

例 8-7　lambda 表达式声明的匿名函数作为列表或字典的元素。

```
1  power = [lambda x: x**2, lambda x: x**3, lambda x: x**4]
2  print(power[0](2), power[1](2), power[2](2))
3  dict1 = {1:lambda x:print(x), 2: lambda x = '扣丁学堂':print(x)}
4  dict1[1]('千锋教育')
5  dict1[2]()
```

运行结果如图 8.7 所示。

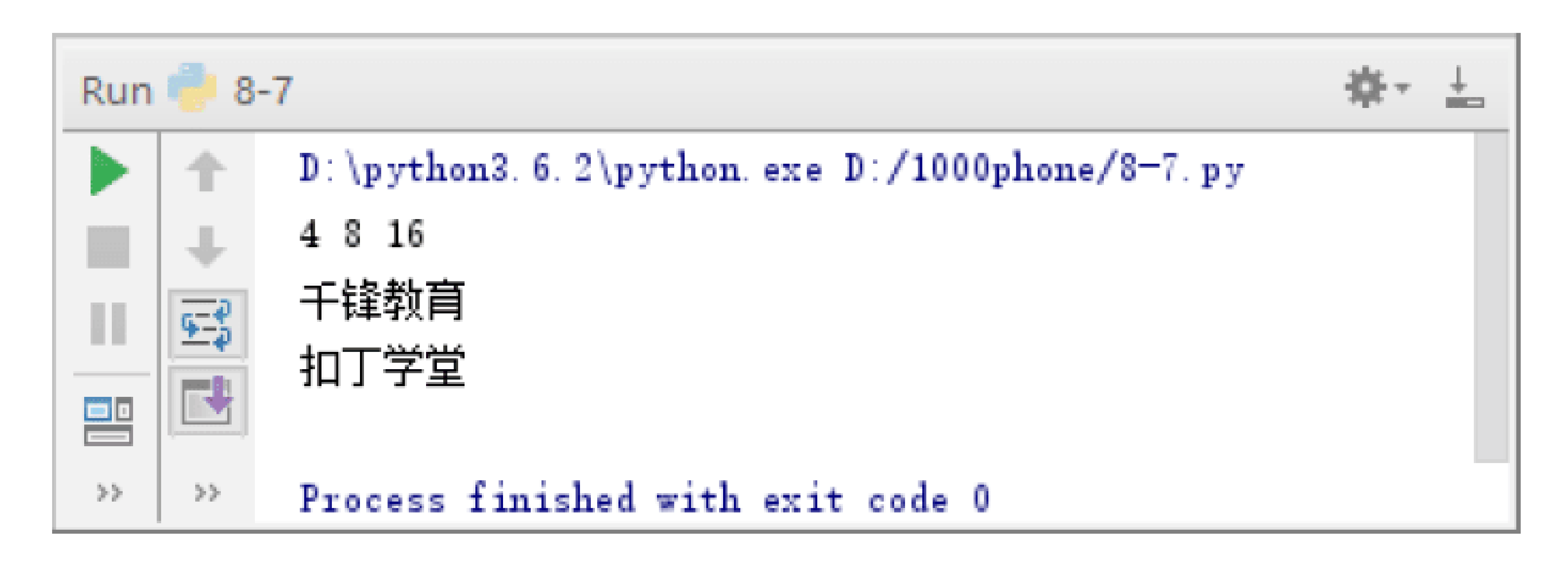


图 8.7　例 8-7 运行结果

在例 8-7 中，第 1 行列表 power 中的每个元素都为一个 lambda 表达式，即构成一个匿名函数列表。第 2 行分别调用列表 power 中的每个匿名函数。第 3 行字典 dict1 中键对应的值为 lambda 表达式，注意 lambda 表达式中也可以含有默认参数。第 4 行调用字典 dict1 中键为 1 对应的匿名函数。第 5 行调用字典 dict1 中键为 2 对应的匿名函数，此处使用默认参数。

8.3　闭　　包

在前面章节中，函数可以通过 return 返回一个变量。此外，函数也可以返回另外一个函数名，如例 8-8 所示。

例 8-8　一个函数返回另外一个函数名。

```
def f1():
    print('f1()函数')
def f2():
    print('f2()函数')
    return f1
x = f2()
x()
f1()  # 正确
```

运行结果如图 8.8 所示。

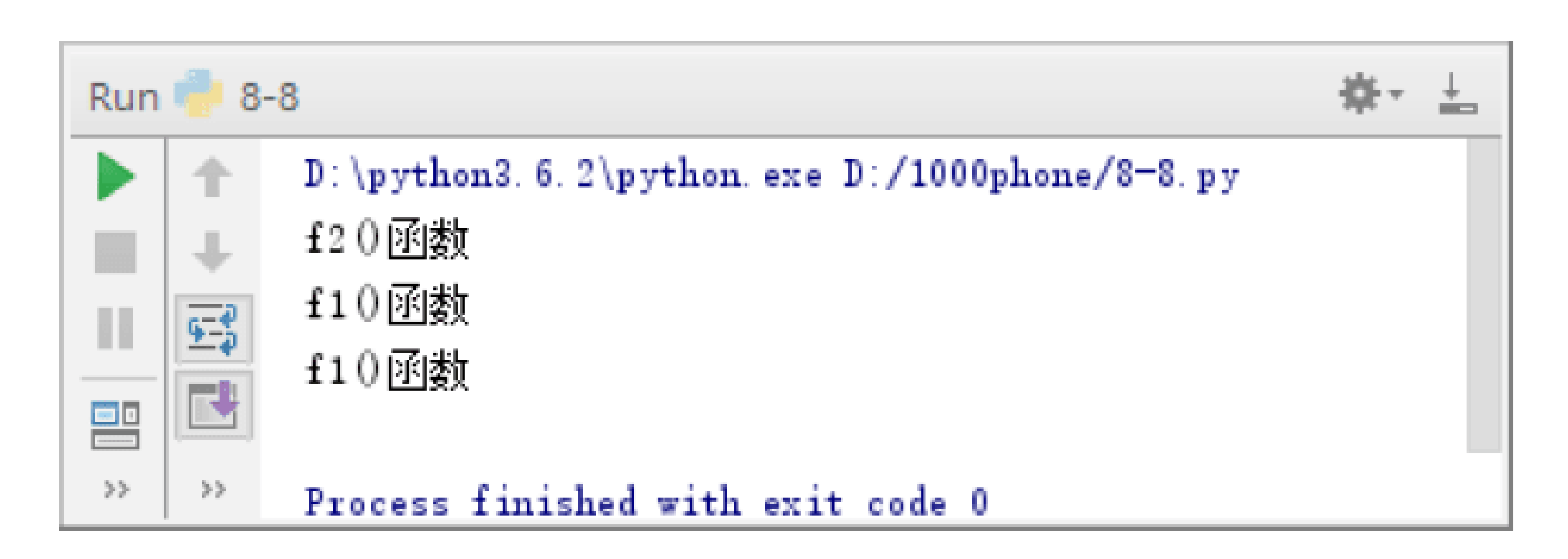


图 8.8 例 8-8 运行结果

在例 8-8 中，第 5 行在函数 f2()中返回一个函数名 f1，第 6 行调用 f2()函数并将返回值赋值给 x，第 7 行通过变量 x 间接调用 f1()函数。

此外，还可以将 f1()函数的定义移动到 f2()函数中，在这样 f2()函数外的作用域就不能直接调用 f1()函数，如例 8-9 所示。

例 8-9 将 f1()函数的定义移动到 f2()函数中。

```
1  def f2():
2      print('f2()函数')
3      def f1():
4          print('f1()函数')
5      return f1
6  x = f2()
7  x()
8  # f1()  错误
```

运行结果如图 8.9 所示。

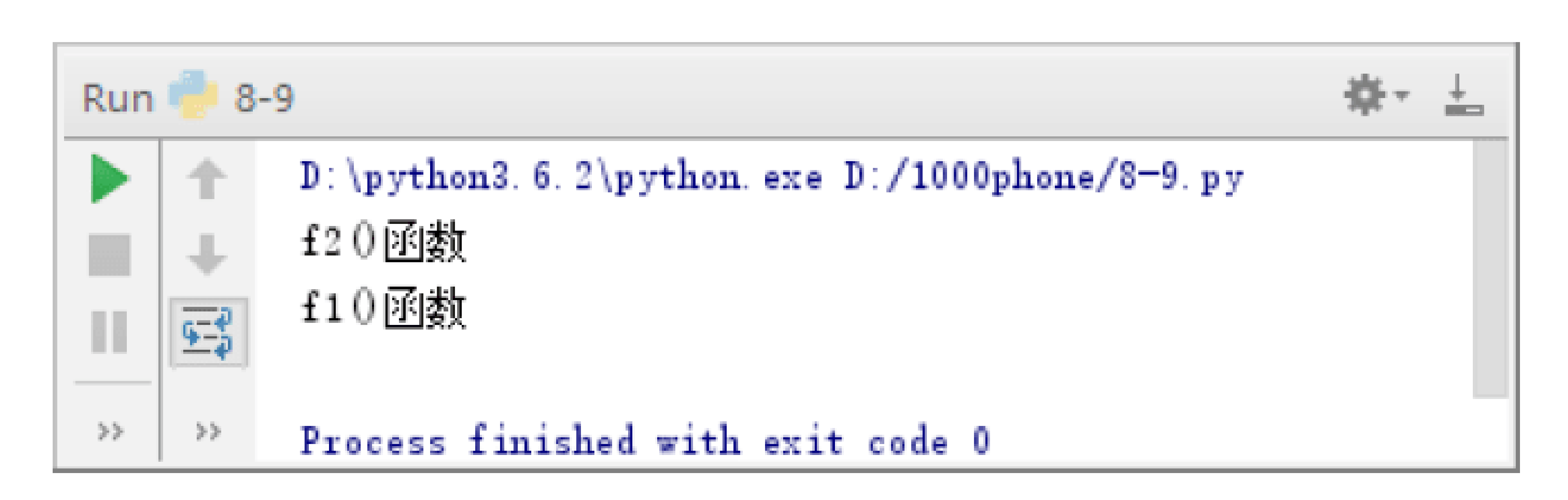


图 8.9 例 8-9 运行结果

在例 8-9 中，函数 f1()的定义嵌套在函数 f2()中，此时在函数 f2()外的作用域不能直接调用函数 f1()，因此将第 8 行代码注释掉。

将一个函数的定义嵌套到另一个函数中，还有其他的作用，如例 8-10 所示。

例 8-10 闭包。

```
list1 = [1, 2, 3, 4]
def f2(list):
    def f1():
        return sum(list)
    return f1
```

```
6    x = f2(list1)
7    print(x())
```

运行结果如图 8.10 所示。

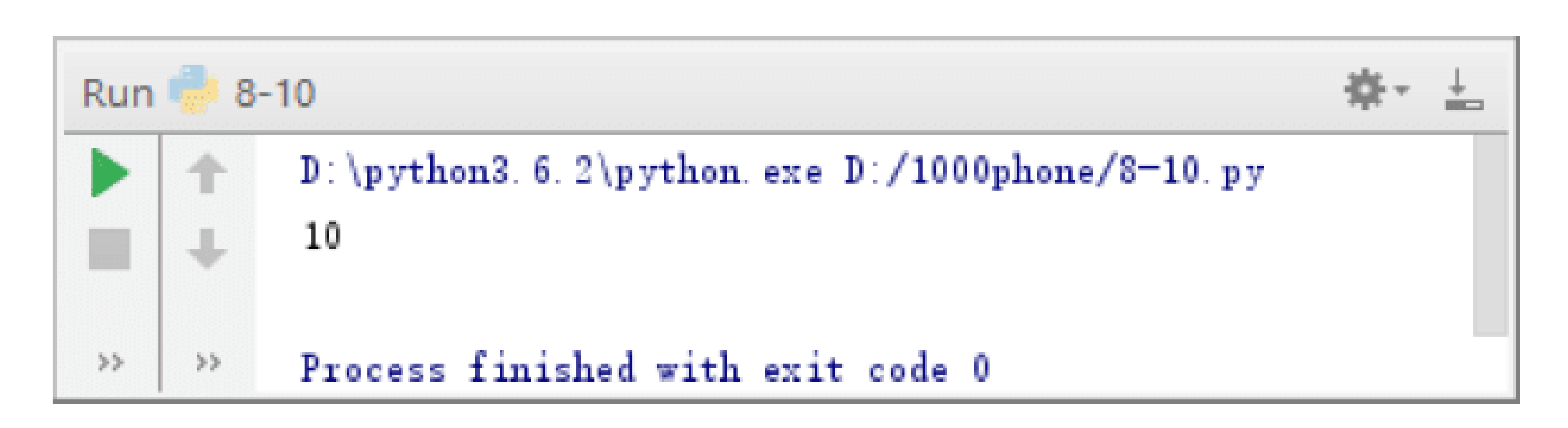


图 8.10　例 8-10 运行结果

在例 8-10 中，函数 f2()中传入一个参数，在函数 f1()中对该参数中的元素求和，具体执行过程如图 8.11 所示。

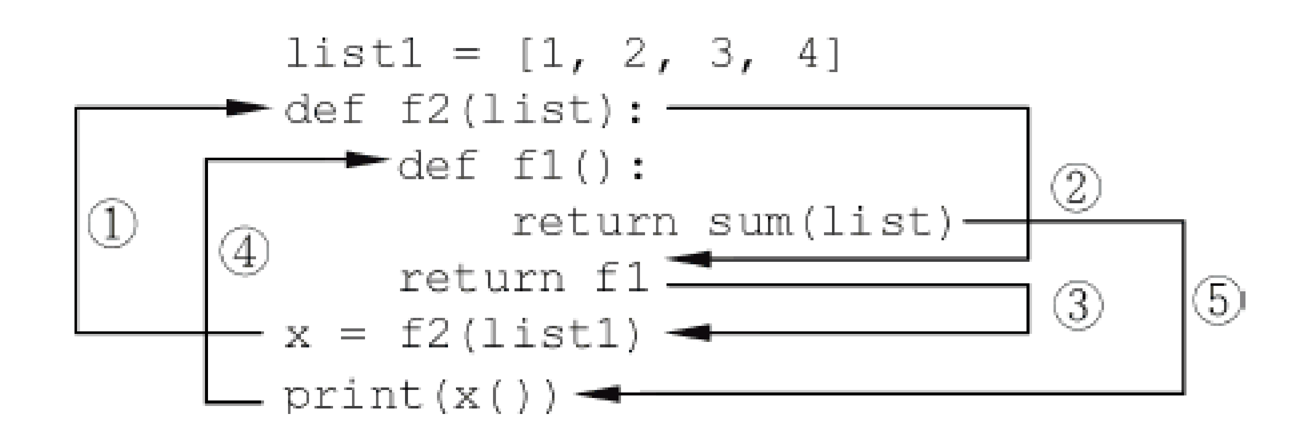


图 8.11　程序执行过程

在图 8.11 中，list1 作为参数传进函数 f2()中，此时不能把函数 f1()移到函数 f2()的外面。因为函数 f1()的功能是计算 list 中所有元素值的和，所以 f1()函数必须依赖于函数 f2()的参数。如果函数 f1()在函数 f2()外，则无法取得 f2()中的数据进行计算，这就引出了闭包的概念。

如果内层函数引用了外层函数的变量（包括其参数），并且外层函数返回内层函数名，这种函数架构称为闭包。从概念中可以得出，闭包需要满足如下 3 个条件：

- 内层函数的定义嵌套在外层函数中。
- 内层函数引用外层函数的变量。
- 外层函数返回内层函数名。

8.4　装　饰　器

在夏天天气晴朗时，人们通常只穿 T 恤就可以了，但当刮风下雨时，人们通常在 T 恤的基础上再增加一件外套，它可以遮风挡雨，并且不影响 T 恤原有的作用，这就是现实生活中装饰器的概念。

8.4.1　装饰器的概念

装饰器本质上还是函数，可以让其他函数在不做任何代码修改的前提下增加额外功

能。它通常用于有切面需求的场景，例如，插入日志、性能测试、权限校验等。

在讲解装饰器之前，先看一段简单的程序，如例 8-11 所示。

例 8-11 将 func()函数的返回值加 1。

```
1   def f2(func):
2       def f1():
3           x = func()
4           return x + 1
5       return f1
6   def func():
7       print('func()函数')
8       return 1
9   decorated = f2(func)
10  print(decorated())
11  print(func())
```

运行结果如图 8.12 所示。

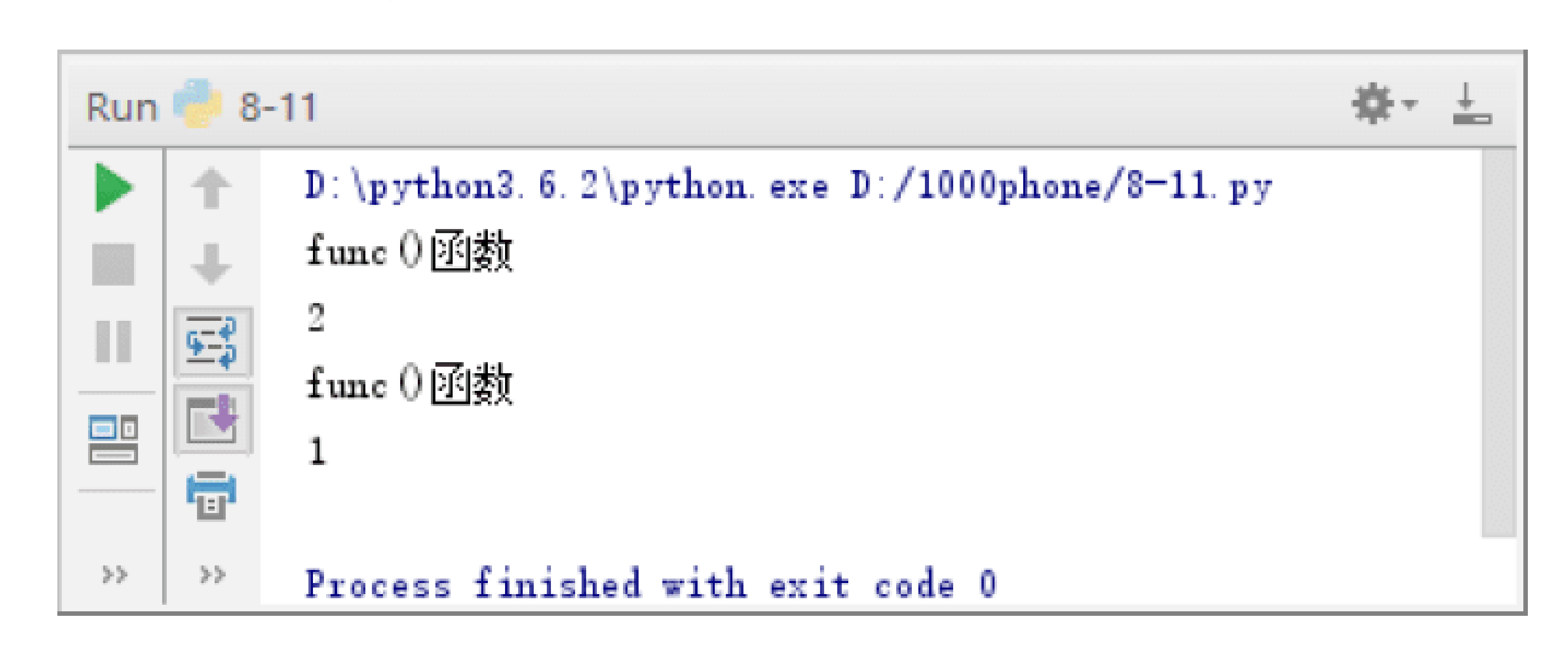


图 8.12 例 8-11 运行结果

在例 8-11 中，第 1 行定义了一个带单个参数 func 的名称为 f2 的函数，第 2 行 f1()函数为闭包的功能函数，其中调用了 func()函数并将 func()函数的返回值加 1 并返回。这样每次 f2()函数被调用时，func 的值可能会不同，但不论 func()代表何种函数，程序都将调用它。

从程序运行结果可看出，调用函数 decorated()的返回值为 2，调用 func()函数的返回值为 1，两者都输出“func()函数”，此时称变量 decorated 是 func 的装饰版，即在 func()函数的基础上增加新功能，本例是将 func()函数的返回值加 1。

还可以用装饰版来“代替”func，这样每次调用时就总能得到“附带其他功能”的 func 版本，如例 8-12 所示。

例 8-12 用装饰版来“代替”func。

```
def f2(func):
    def f1():
        return func() + 1
```

```
4       return f1
5   def func():
6       print('func()函数')
7       return 1
8   func = f2(func)
9   print(func())
```

运行结果如图 8.13 所示。

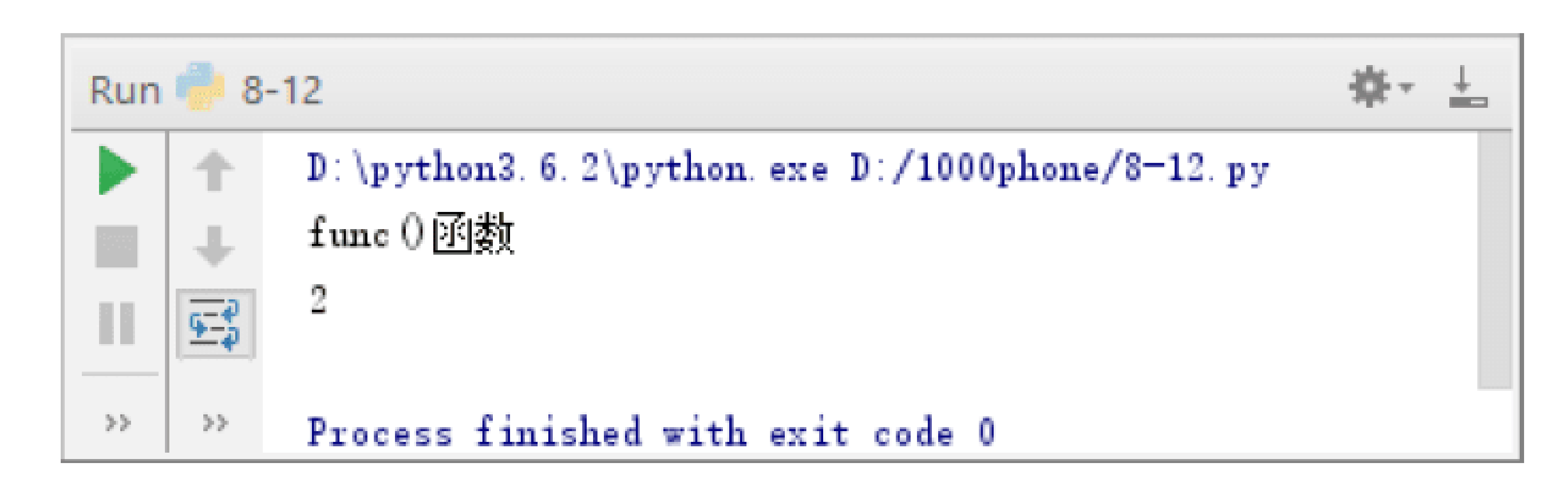


图 8.13　例 8-12 运行结果

在例 8-12 中，第 3 行等价于例 8-11 中的第 3、4 行，第 8 行将例 8-11 中的第 9 行 decorated 改为 func，这样每次通过函数名 func 调用函数时，都将执行装饰后的版本。

通过上例可以得出装饰器的概念，即一个以函数作为参数并返回一个替换函数的可执行函数。装饰器的本质是一个嵌套函数，外层函数的参数是被修饰的函数，内层函数是一个闭包并在其中增加新功能（装饰器的功能函数）。

8.4.2　@符号的应用

例 8-12 中使用变量名将装饰器函数与被装饰函数联系起来。此外，还可以通过@符号和装饰器名实现两者的联系，如例 8-13 所示。

例 8-13　@符号的应用。

```
1   def f2(func):
2       def f1():
3           return func() + 1
4       return f1
5   @f2
6   def func():
7       print('func()函数')
8       return 1
9   print(func())
```

运行结果如图 8.14 所示。

在例 8-13 中，第 5 行通过@符号和装饰器名实现装饰器函数与被装饰函数联系。第 9 行调用 func()函数时，程序会自动调用装饰器函数的代码。

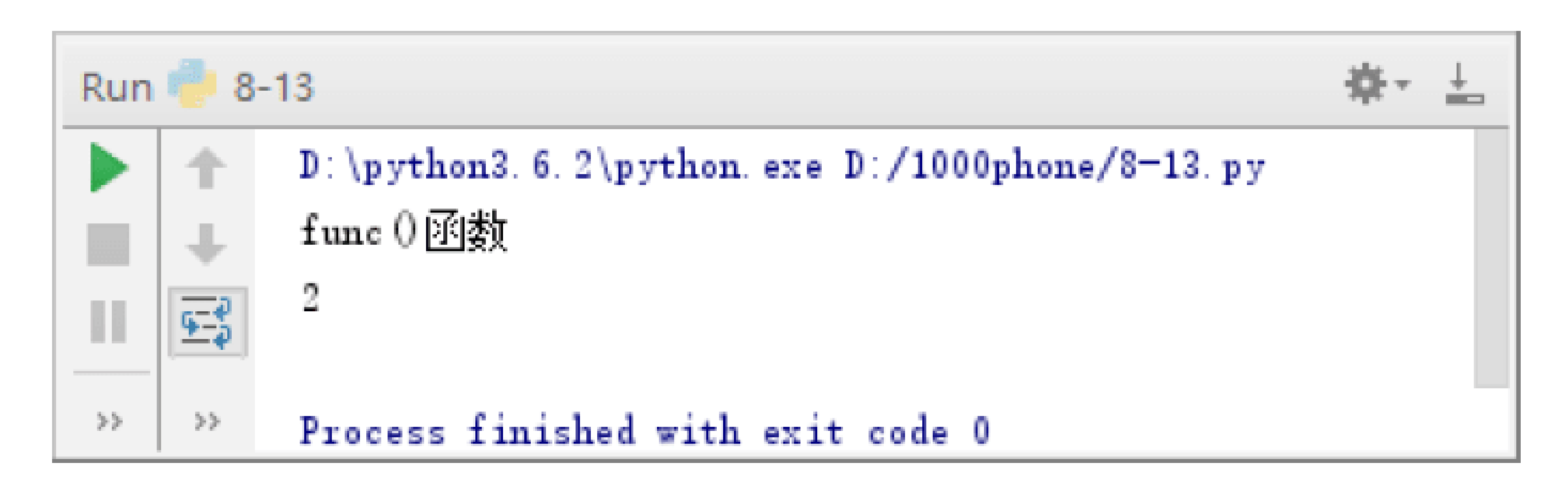


图 8.14　例 8-13 运行结果

8.4.3　装饰有参数的函数

装饰器除了可以装饰无参数的函数外，还可以装饰有参数的函数，如例 8-14 所示。

例 8-14　用装饰器装饰有参数的函数。

```
1  def f2(func):
2      def f1(a = 0, b = 0):
3          return func(a, b) + 1
4      return f1
5  @f2
6  def func(a = 0, b = 0):
7      print('func()函数')
8      return a + b
9  print(func(2, 3))
```

运行结果如图 8.15 所示。

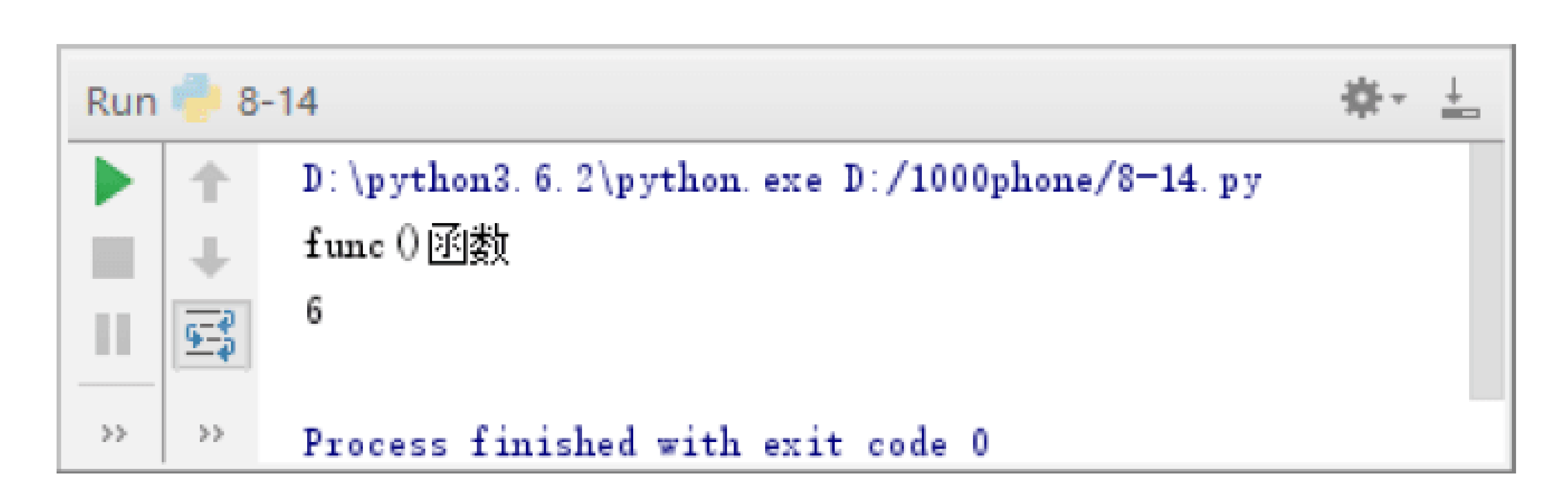


图 8.15　例 8-14 运行结果

在例 8-14 中，第 6 行定义一个带有两个默认参数的 func()函数。第 5 行将 f2()函数声明为装饰器函数，用来修饰 func()函数。第 9 行调用 func 装饰器函数，注意 f1()函数中的参数必须包含对应 func()函数的参数。

8.4.4　带参数的装饰器——装饰器工厂

通过上面的学习可知，装饰器本身也是一个函数，即装饰器本身也可以带参数，此时装饰器需要再多一层内嵌函数，如例 8-15 所示。

例 8-15 带参数的装饰器。

```
1   def f3(arg = '装饰器的参数'):
2       def f2(func):
3           def f1():
4               print(arg)
5               return func() + 1
6           return f1
7       return f2
8   @f3('带参数的装饰器')
9   def func():
10      print('func()函数')
11      return 1
12  print(func())
```

运行结果如图 8.16 所示。



图 8.16 例 8-15 运行结果

在例 8-15 中，第 1 行定义装饰器函数，其由 3 个函数嵌套而成，最外层函数有一个装饰器自带的参数，内层函数不变，相当于闭包的嵌套。第 8 行将 f3()函数声明为装饰器函数，用来修饰 func()函数。

若大家不理解此代码，可以将装饰器写成如下代码，如例 8-16 所示。

例 8-16 装饰器分解成闭包的嵌套。

```
def f3(arg = '装饰器的参数'):
    def f2(func):
        def f1():
            print(arg)
            return func() + 1
        return f1
    return f2
def func():
    print('func()函数')
    return 1
f2 = f3('带参数的装饰器')
func = f2(func)
print(func())
```

运行结果如图 8.17 所示。

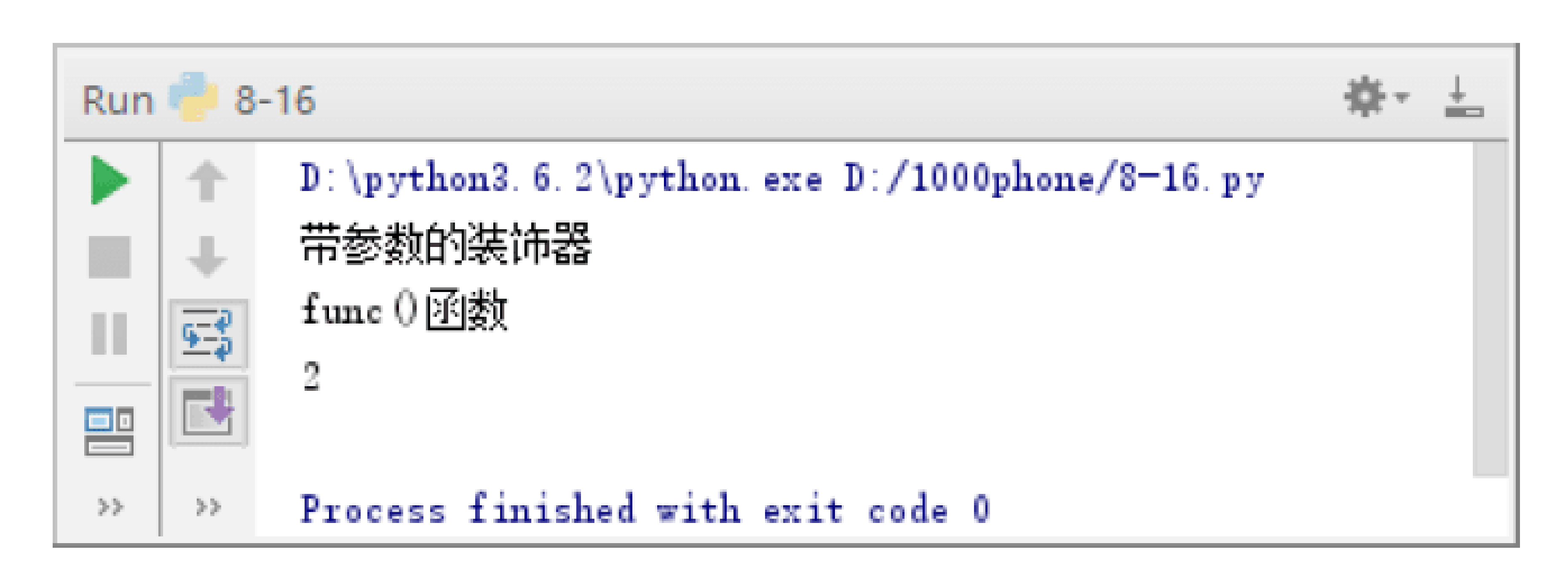


图 8.17 例 8-16 运行结果

在例 8-16 中，将装饰器分解成闭包的嵌套，这种写法更容易理解。此外，还可以将第 11、12 行代码写成如下代码：

```
func = f3('带参数的装饰器')(func)
```

上述代码相当于省略中间变量 f2。

8.5 偏 函 数

函数最重要的一个功能的是复用代码，有时在复用已有函数时，可能需要固定其中的部分参数，除了设置默认值参数外，还可以使用偏函数（用来固定函数调用时部分或全部参数的函数叫偏函数），如例 8-17 所示。

例 8-17 偏函数。

```
1  def myAdd1(a, b, c):
2      return a + b + c
3  def myAdd2(a, b):
4      return myAdd1(a, b, 123)
5  print(myAdd2(1, 1))
```

运行结果如图 8.18 所示。

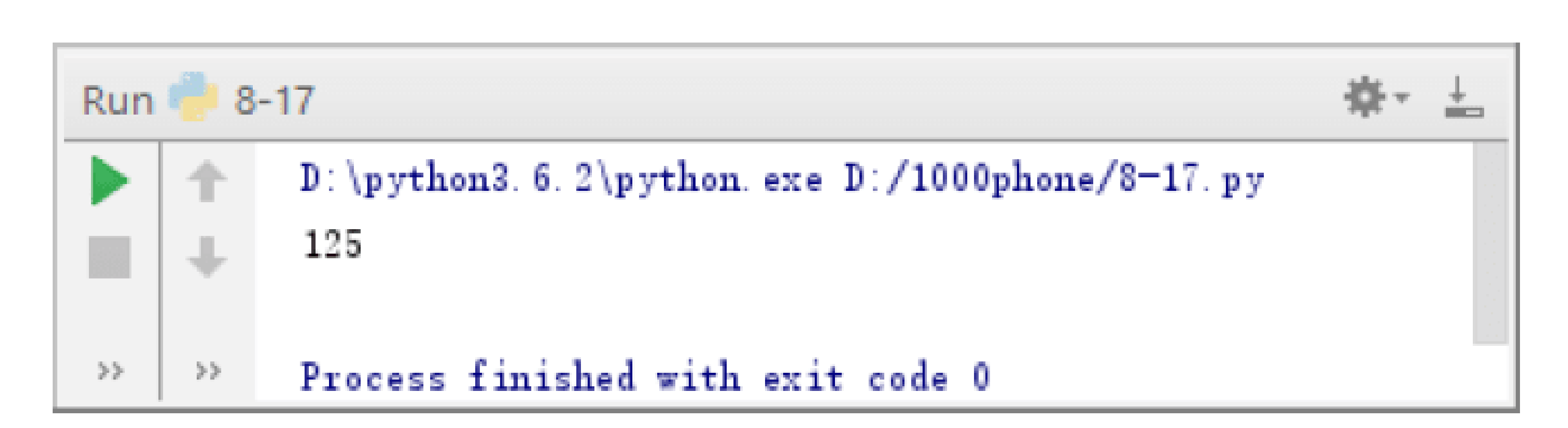


图 8.18 例 8-17 运行结果

在例 8-17 中，第 3 行定义一个 myAdd2()函数，与第 1 行 myAdd1()函数的区别仅在于参数 c 固定为一个数字 123，这时就可以使用偏函数来复用上面的函数。

8.6 常用的内建函数

在 Python 中，内建函数是被自动加载的，可以随时调用这些函数，不需要定义，极大地简化了编程。

8.6.1 eval()函数

eval()函数用于对动态表达式求值，其语法格式如下：

```
eval(source, globals = None, locals = None)
```

其中，source 是动态表达式的字符串，globals 和 locals 是求值时使用的上下文环境的全局变量和局部变量，如果不指定，则使用当前运行上下文。

接下来演示 eval()函数的用法，如例 8-18 所示。

例 8-18 eval()函数的用法。

```
x = 3
str = input('请输入包含 x（x = 3）的 Python 表达式：')
print(str, '的结果为', eval(str))
```

运行结果如图 8.19 所示。

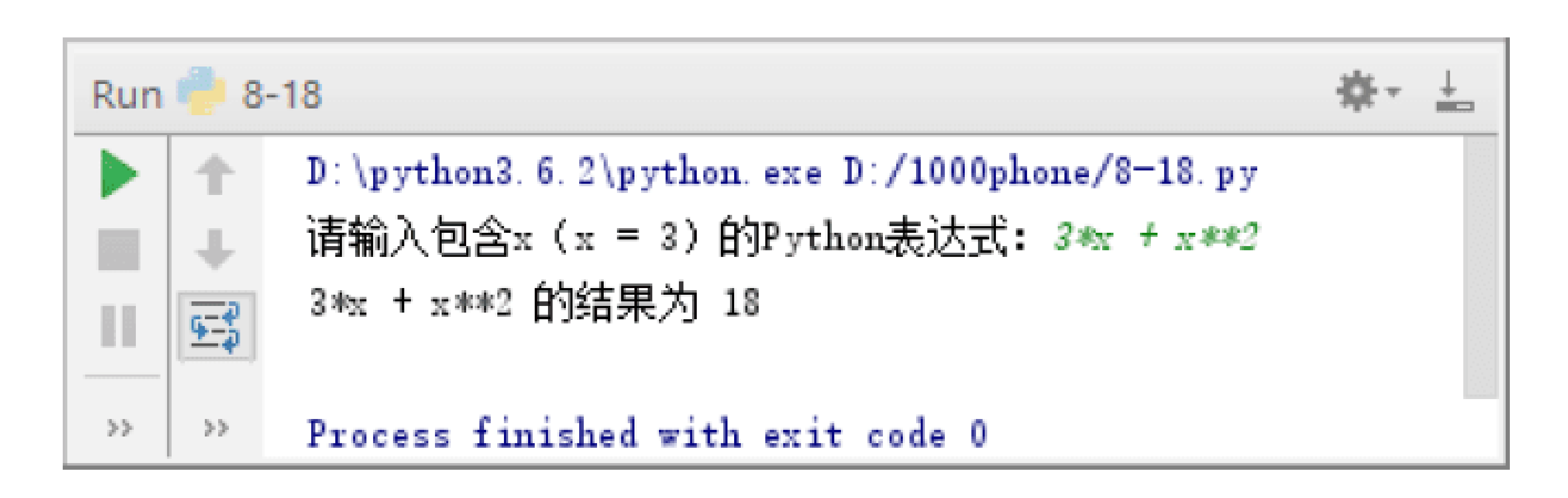


图 8.19 例 8-18 运行结果

在例 8-18 中，通过 input()函数输入 Python 表达式，接着通过 eval()函数求出该表达式的值。

8.6.2 exec()函数

exec()函数用于动态语句的执行，其语法格式如下：

```
exec(source, globals = None, locals = None)
```

其中，source 是动态语句的字符串，globals 和 locals 是使用的上下文环境的全局变量和

局部变量，如果不指定，则使用当前运行上下文。

接下来演示 exec()函数的用法，如例 8-19 所示。

例 8-19 exec()函数的用法。

```
str = input('请输入 Python 语句：')
exec(str)
```

运行结果如图 8.20 所示。

```
Run 8-19
D:\python3.6.2\python.exe D:/1000phone/8-19.py
请输入Python语句：for i in range(4):print(i)
0
1
2
3

Process finished with exit code 0
```

图 8.20 例 8-19 运行结果

在例 8-19 中，通过 input()函数输入 Python 语句，接着通过 exec()函数执行该语句。

8.6.3 compile()函数

compile()函数用于将一个字符串编译为字节代码，其语法格式如下：

```
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)
```

其中，source 为代码语句的字符串，filename 为代码文件名称，如果不是从文件读取代码，则传递一些可辨认的值，mode 为指定编译代码的种类，其值可以为'exec'、'eval'、'single'，剩余参数一般使用默认值。

接下来演示 compile()函数的用法，如例 8-20 所示。

例 8-20 compile()函数的用法。

```
str = input('请输入 Python 语句：')
co = compile(str, '', 'exec')
exec(co)
```

运行结果如图 8.21 所示。

在例 8-20 中，通过 input()函数输入 Python 语句，接着通过 compile()函数将字符串 str 转换为字节代码对象。

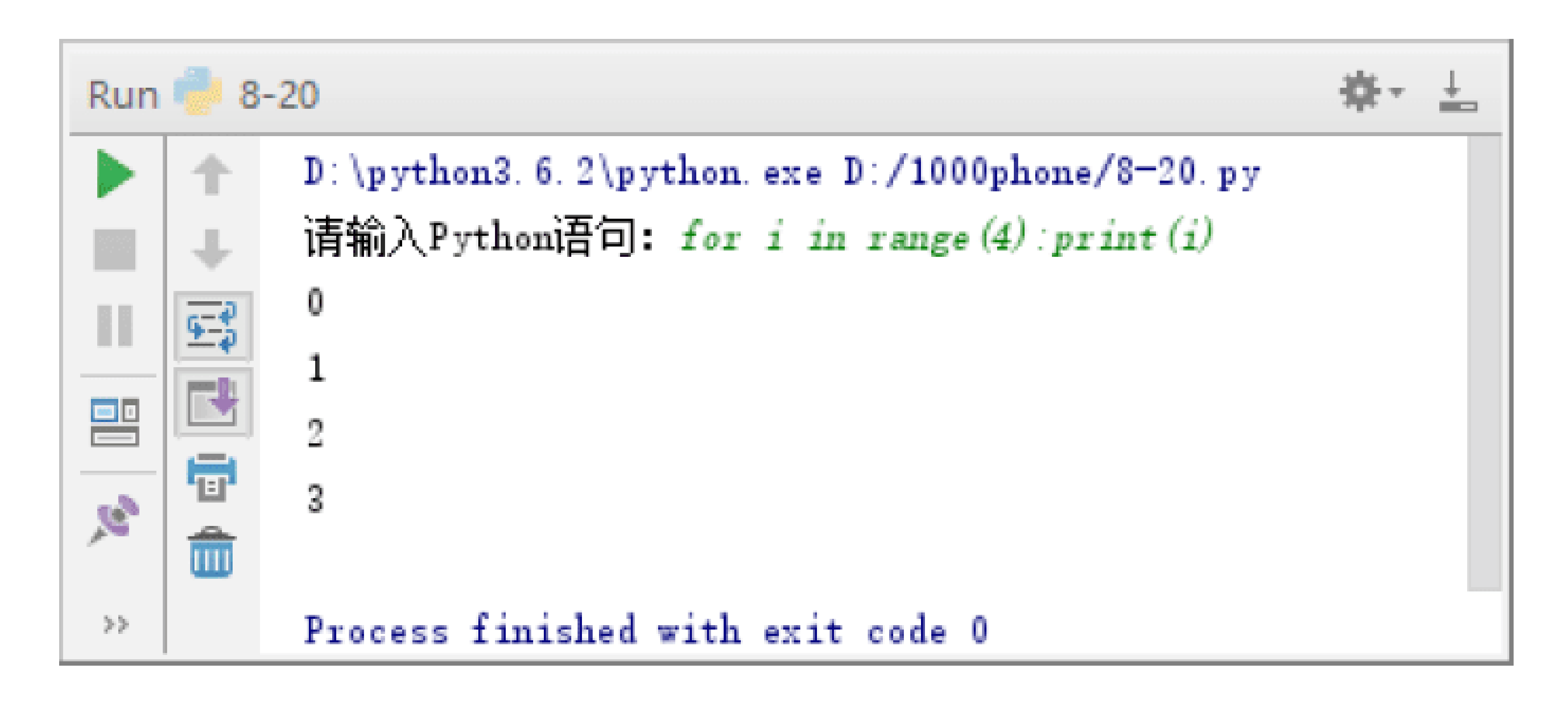


图 8.21　例 8-20 运行结果

8.6.4　map()函数

程序中经常需要对列表和其他序列中的每个元素进行同一个操作并将其结果集合起来，具体示例如下：

```
list1, list2 = [1, 2, 3, 4], []
for i in list1:
    list2.append(i + 10)
print(list2)
```

上述代码表示将 list1 中的每个元素加 10 并添加到 list2 中。该程序运行后，输出结果如下：

```
[11, 12, 13, 14]
```

实际上，Python 提供了一个更方便的工具来完成此种操作，这就是 map()函数，其语法格式如下：

```
map(function, sequence[, sequence,…])
```

其中，function 为函数名，其余参数为序列，返回值为迭代器对象，通过 list()函数可以将其转换为列表，也可以使用 for 循环进行遍历操作。

接下来演示 map()函数的用法，如例 8-21 所示。

例 8-21　map()函数的用法。

```
list1 = [1, 2, 3, 4]
func = lambda x : x + 10
list2 = list(map(func, list1))
print(list2)
```

运行结果如图 8.22 所示。

在例 8-21 中，map()函数对列表 list1 中的每个元素调用 func 函数并将返回结果组成一个可迭代对象，如图 8.23 所示。

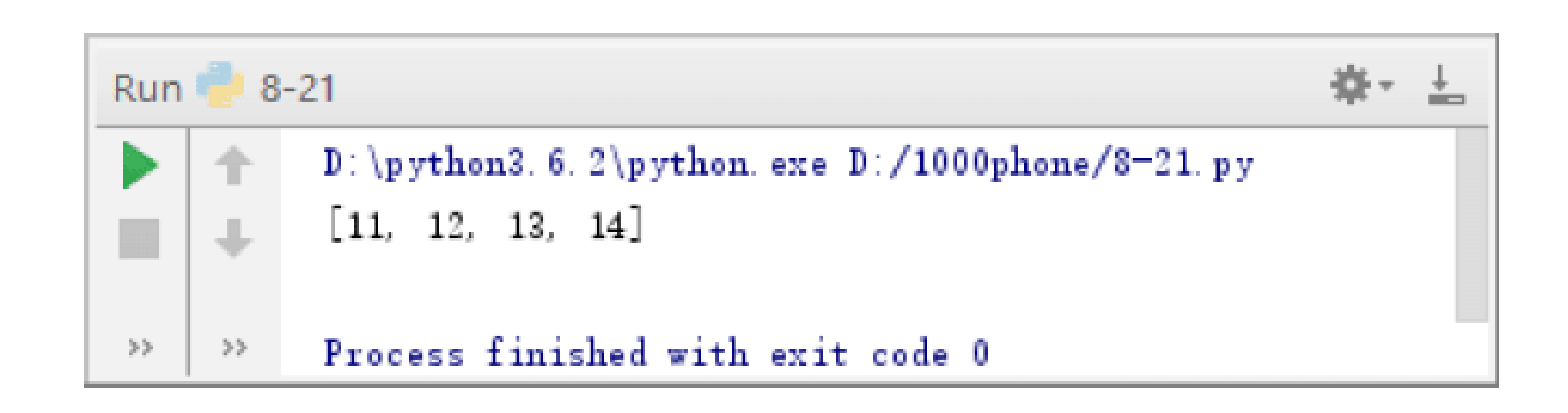


图 8.22 例 8-21 运行结果

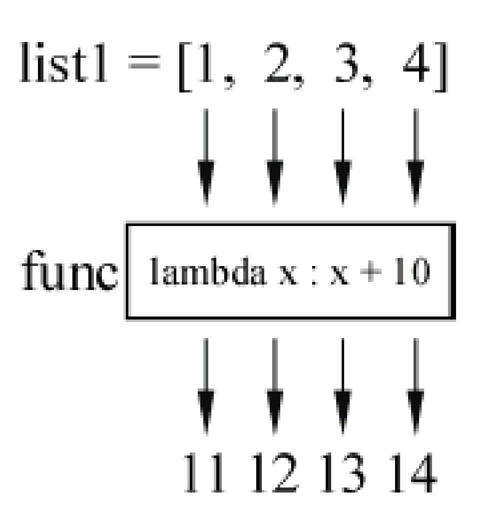


图 8.23 map()函数执行过程

此外，map()函数还可以接收两个序列，具体示例如下：

```
list = list(map(lambda x, y : x + y, range(1, 5), range(5, 9)))
print(list)
```

该程序运行后，输出结果如下：

```
[6, 8, 10, 12]
```

8.6.5 filter()函数

filter()函数可以对指定序列进行过滤操作，其语法格式如下：

```
filter(function, sequence)
```

其中，function 为函数名，它所引用的函数只能接收一个参数，并且返回值是布尔值，sequence 为一个序列，filter()函数返回值为迭代器对象。

接下来演示 filter()函数的用法，如例 8-22 所示。

例 8-22 filter()函数的用法。

```
seq = ['qianfeng', 'codingke.com', '*#$']
func = lambda x : x.isalnum()
list = list(filter(func, seq))
print(list)
```

运行结果如图 8.24 所示。

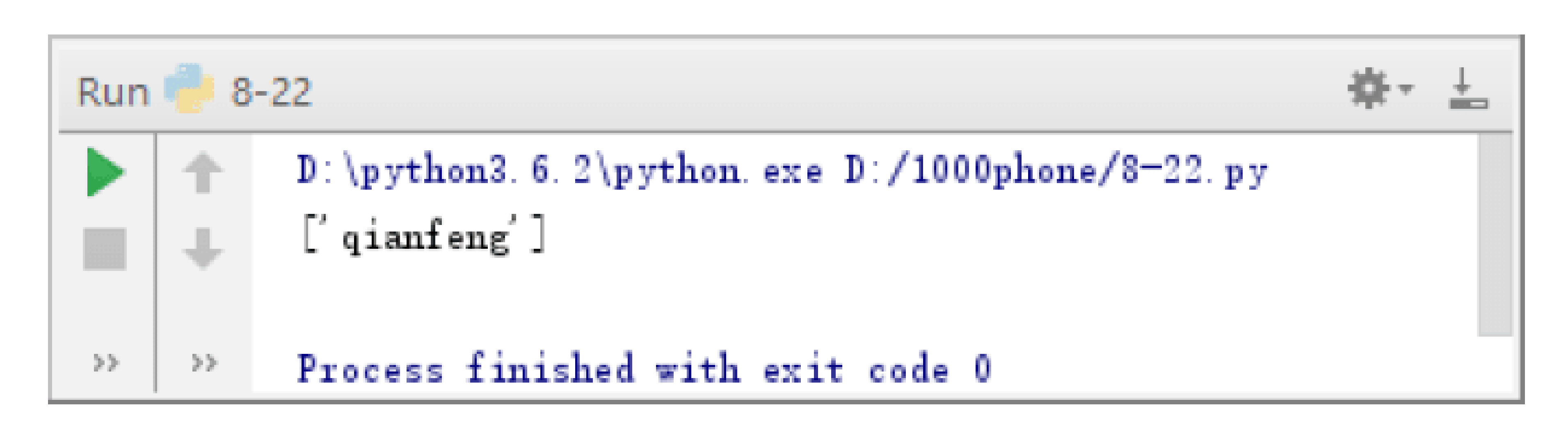


图 8.24 例 8-22 运行结果

在例 8-22 中，filter()函数对列表 list 中的每个元素调用 func 函数并返回使得 func 函数返回值为 True 的元素组成的可迭代对象，如图 8.25 所示。

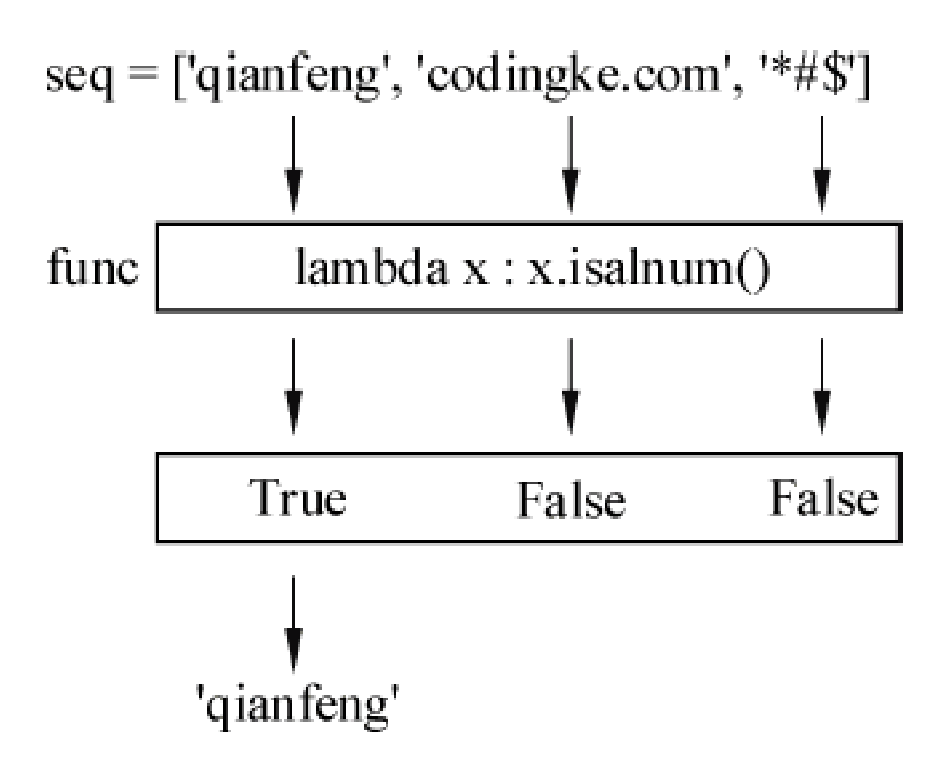


图 8.25　filter()函数执行过程

8.6.6　zip()函数

zip()函数用于将一系列可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的迭代对象，如例 8-23 所示。

例 8-23　zip()函数的用法。

```
list1, list2 = [1, 2, 3], ['千锋教育', '扣丁学堂', '好程序员特训营']
list3 = list(zip(list1, list2))
print(list3)
```

运行结果如图 8.26 所示。

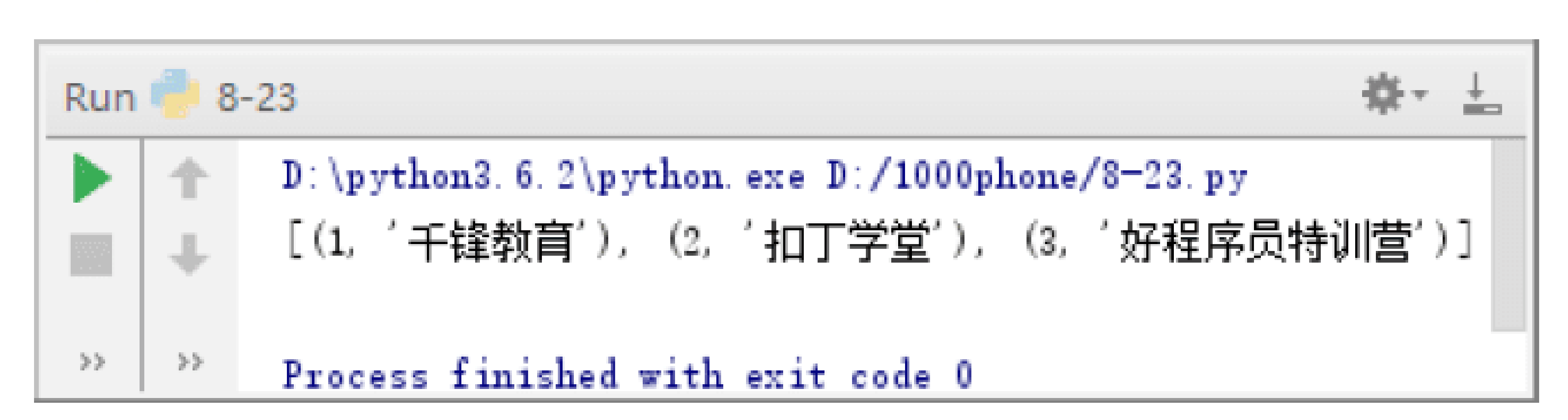


图 8.26　例 8-23 运行结果

在例 8-23 中，zip()函数将列表 list1 中第 1 个元素与列表 list2 中的 1 个元素组成一个元组，以此类推，最终返回由 3 个元组组成的迭代对象。

zip()参数可以接受任何类型的序列，同时也可以有两个以上的参数。但当传入参数的长度不同时，zip()函数以最短序列长度为准进行截取获得元组，具体示例如下：

```
list1, list2, list3 = [1, 2, 3, 4], ['a', {1, 'a'}], [3.4, 5]
list4 = list(zip(list1, list2, list3))
print(list4)
```

该程序运行后，输出结果如下：

```
[(1, 'a', 3.4), (2, {1, 'a'}, 5)]
```

此外，在 zip()函数中还可以使用*运算符，如例 8-24 所示。

例 8-24　在 zip()函数中使用*运算符。

```
list1, list2 = [1, 2, 3], ['千锋教育', '扣丁学堂', '好程序员特训营']
```

```
2    zipped = zip(list1, list2)
3    list4 = zip(*zipped)
4    print(list(list4))
```

运行结果如图 8.27 所示。

```
Run 8-24
D:\python3.6.2\python.exe D:/1000phone/8-24.py
[(1, 2, 3), ('千锋教育', '扣丁学堂', '好程序员特训营')]

Process finished with exit code 0
```

图 8.27　例 8-24 运行结果

在例 8-24 中，第 3 行 zip()函数中使用*运算符相当于执行相反的操作。

在 Python 中，还有许多内建函数，当要用到某个函数时，只需在 PyCharm 编辑器中写出函数名，它就会自动提示函数的参数。例如，在编辑器中输入 map 后出现如图 8.28 所示的提示。

```
f map(func, iter1, iter2)                                    builtins
Press Ctrl+句点 to choose the selected (or first) suggestion and insert a dot afterwards >>
```

图 8.28　函数参数提示

此时在编辑器中接着输入()，则会提示函数的每个参数类型，如图 8.29 所示。

```
func: (TypeVar('_T1'), TypeVar('_T2')) -> TypeVar('_S'),
iter1: Iterable[TypeVar('_T1')], iter2: Iterable[TypeVar('_T2')]

func: (TypeVar('_T1')) -> TypeVar('_S'), iter1: Iterable[TypeVar('_T1')]
```

图 8.29　参数类型提示

8.7 小 案 例

8.7.1 案例一

假设已实现用户聊天、购买商品、显示个人信息等功能，在使用这些功能前需验证用户使用的登录方式（微信、QQ 或其他）及身份信息，要求使用装饰器实现该功能，具体实现如例 8-25 所示。

例 8-25　装饰器案例。

```
type = input('请输入登录方式: ')
status = False
name, pwd= "小千", "123"  # 假设从数据库中获取到用户信息
```

```
def login(checkType):
    def check(func):
        def wrapper(*args, **kwargs):
            if checkType == 'WeChat'or checkType == 'QQ':
                global status
                if status == False:
                    username = input("user:")
                    password = input("password:")
                    if username == name and password == pwd:
                        print("欢迎%s! "%name)
                        status = True
                    else:
                        print("用户名或密码错误! ")
                if status == True:
                    func(*args,**kwargs)
            else:
                print('仅支持微信或 QQ 登录! ')
        return wrapper
    return check
@login(type)
def shop():
    print('购物')
@login(type)
def info():
    print('个人信息')
@login(type)
def chat():
    print('聊天')
info()
chat()
shop()
```

运行结果如图 8.30 所示。

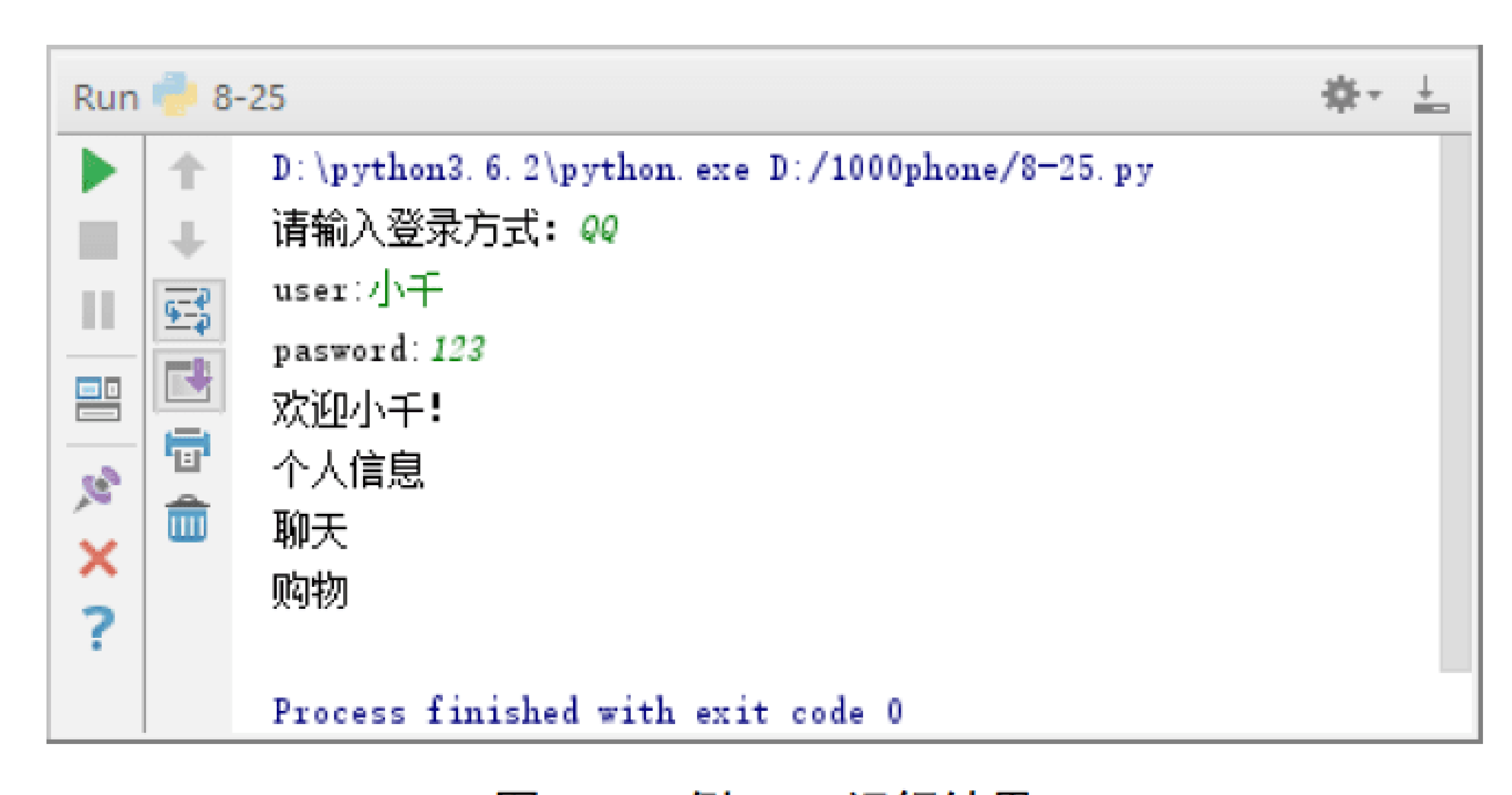


图 8.30　例 8-25 运行结果

在例 8-25 中，使用带参数的装饰器 login()修饰函数 info()、chat()和 shop()，这样在每次调用这 3 个函数时，将变得非常简单。

8.7.2 案例二

若有以下学生信息，如表 8.1 所示。现要求只对男同学的成绩进行由高到低排序并输出排序后学生的姓名与成绩，具体实现如例 8-26 所示。

表 8.1 学生信息

姓 名	性 别	分 数	姓 名	性 别	分 数
小千	女	95	小丁	男	88
小锋	男	99	小明	男	90
小扣	女	86	…	…	…

例 8-26 对男同学的成绩进行由高到低排序并输出排序后的学生姓名与成绩。

```
1   info = [{'name':'小千', 'sex':0, 'score':95},    # 0 代表女性
2          {'name':'小锋', 'sex':1, 'score':99},     # 1 代表男性
3          {'name':'小扣', 'sex':0, 'score':86},
4          {'name':'小丁', 'sex':1, 'score':88},
5          {'name':'小明', 'sex':1, 'score':90}]
6   temp1 = list(filter(lambda x:x['sex'] == 1, info))
7   temp2 = list(sorted(temp1, key = lambda x:x['score'], reverse = True))
8   for x in temp2:
9       for key, value in x.items():
10          if key != 'sex':
11              print(value, end = ' ')
12      print()
```

运行结果如图 8.31 所示。

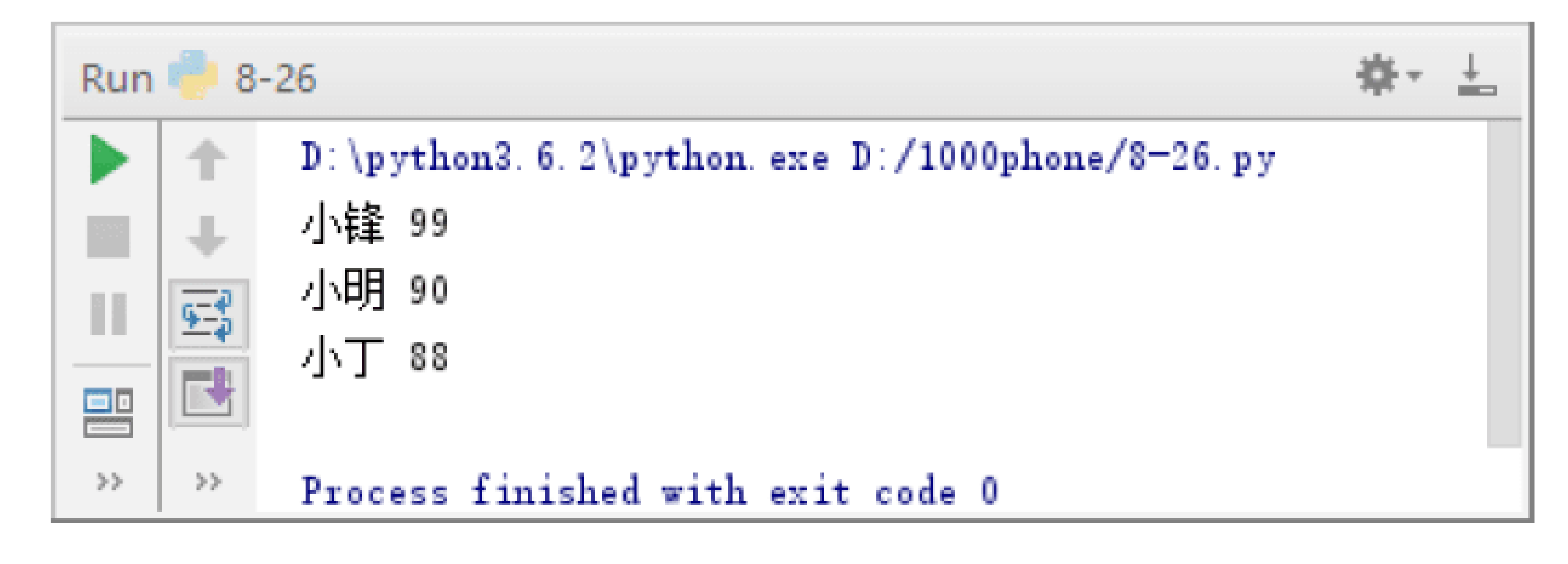


图 8.31 例 8-26 运行结果

在例 8-26 中，第 6 行使用 filter()函数过滤出'sex'为 1 的元素，第 7 行使用 sorted()函数对过滤出的元素再进行降序排序。

8.8 本章小结

本章主要介绍了 Python 中函数的高级用法，包括间接调用函数、匿名函数、闭包、装饰器、偏函数及常用的内建函数。通过本章的学习，应理解闭包及装饰器的用法并能够应用到实际开发中。

8.9 习 题

1．填空题

（1）若一个函数引用另一个函数中的变量，则可以使用________实现。
（2）装饰器本质上是________。
（3）装饰器的内层函数是一个________。
（4）在函数定义前添加装饰器名和________符号实现对函数的装饰。
（5）带参数的装饰器实现时需多一层________函数。

2．选择题

（1）（　　）函数可以对指定序列进行过滤。
　A．map()　B．filter()　C．sorted()　D．zip()
（2）匿名函数可以通过（　　）关键字进行声明。
　A．def　B．return　C．lambda　D．anonymous
（3）程序调用 filter()函数时，第一个参数所引用的函数返回值是（　　）。
　A．布尔值　B．字符串　C．列表值　D．元组值
（4）（　　）函数根据传入的函数对指定序列做操作。
　A．exec()　B．eval()　C．map()　D．zip()
（5）若 print(list(zip(range(2), range(2, 5))))，则输出（　　）。
　A．[(0, 2), (1, 3), (0, 4)]　B．[(2, 0), (3, 1)]
　C．[(1, 3), (0, 4)]　D．[(0, 2), (1, 3)]

3．思考题

（1）简述闭包的概念。
（2）简述装饰器的概念。



4．编程题

编写程序，要求使用两个装饰器装饰同一函数。

第 9 章 chapter 9

模块与包

本章学习目标

- 理解模块与包的概念。
- 掌握模块的导入。
- 熟悉内置标准模块。
- 掌握自定义模块。
- 掌握包的发布与安装。

模块可以将函数按功能划分到一起，以便共享给他人使用。一个 Python 文件就可以视作一个模块，模块提供了将独立文件连接构建更复杂 Python 程序的方式。

9.1 模块的概念

模块是一个保存了 Python 代码的文件，其中可以包含变量、函数或类的定义，也可以包含其他各种 Python 语句。使用模块有以下 3 方面的优势。

（1）模块提高了代码的可维护性。在程序开发过程中，随着程序功能的增多，在一个文件中的代码会越来越长，从而造成程序不易维护，此时可以把相关功能的代码分配到一个模块里，从而使代码更易懂、更易维护。

（2）模块提高了代码的可重用性。在应用程序开发中，经常需要处理时间，此时不必在每个程序中写入时间的处理函数，只需导入 time 模块即可。

（3）模块避免了函数名和变量名冲突。由于相同名字的函数和变量可以分别存在于不同模块中，在编写模块时，不必考虑名字会与其他模块冲突（此处不考虑导包情况）。

在 Python 中，模块可以分为 3 类，具体如下所示：

- 内置标准模块（标准库）——Python 自带的模块，如 sys、os 等。
- 自定义模块——用户为了实现某个功能自己编写的模块。
- 第三方模块——其他人已经编写好的模块。

一个 Python 程序可由若干模块构成，一个模块中可以使用其他模块的变量、函数和类等，如图 9.1 所示。

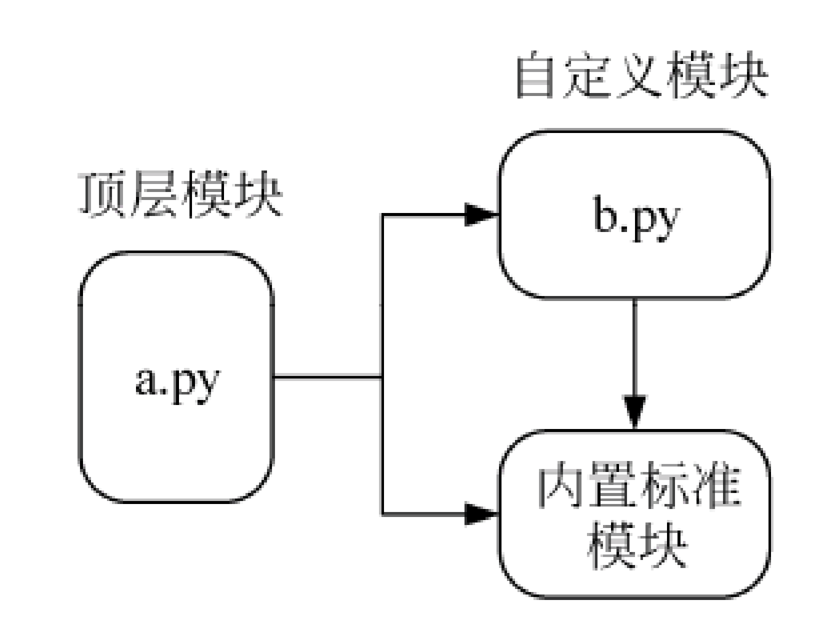


图 9.1 模块原理图

在图 9.1 中， a.py 是一个顶层模块（又称主模块），其中使用了自定义模块 b.py 和内置标准模块，b.py 也使用了内置标准模块。

接下来简单演示模块的使用，如例 9-1 所示。

例 9-1 模块的使用。

```
import math                # 导入内置标准模块 math
num = math.sqrt(9)         # 使用 math 模块中的 sqrt()函数
print(num)
```

运行结果如图 9.2 所示。

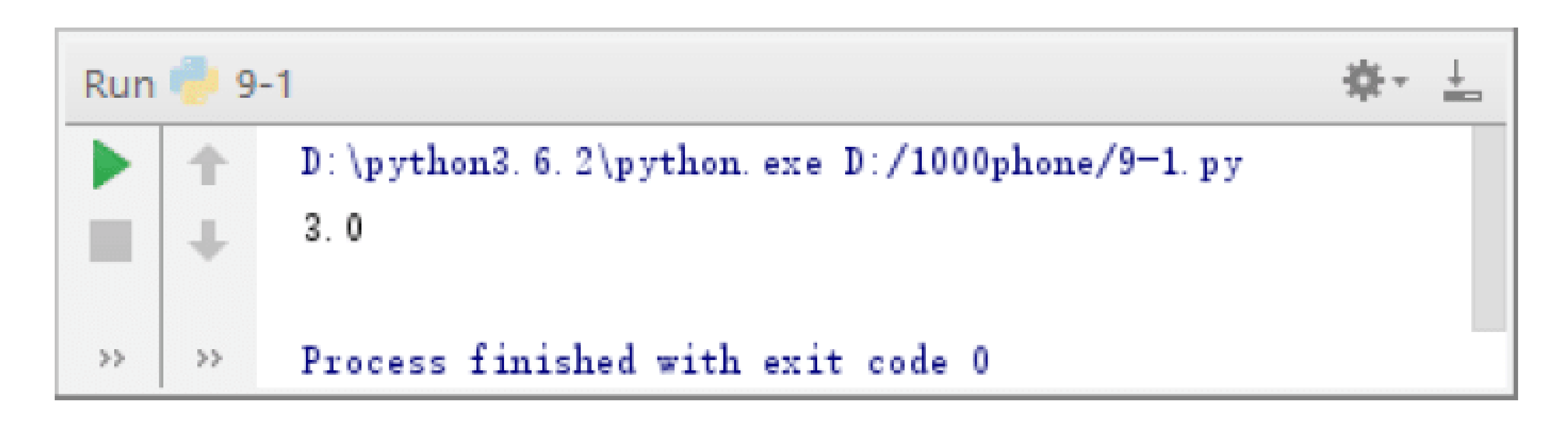


图 9.2 例 9-1 运行结果

在例 9-1 中，程序使用 math 模板中的 sqrt()函数可以很方便地计算出 9 的算术平方根。

9.2 模块的导入

模块需要先导入，然后才能使用其中的变量或函数。在 Python 中使用关键字 import 导入某个模块，其语法格式如下：

```
import 模块名                       # 导入模块
import 模块名 1, 模块名 2, …        # 导入多个模块
import 模块名 as 别名               # 为模块指定别名
```

其中，import 用于导入整个模块，可用 as 为导入的模块指定一个别名。使用 import 导入模块后，模块中的对象均以“模块名（别名）.对象名称”的方式来引用。

接下来演示 import 关键字导入模块，如例 9-2 所示。

例 9-2 import 关键字导入模块。

```
1  import math as m
2  import sys, time
3  print(m.sqrt(9))
4  print(sys.platform)
5  print(time.time())
```

运行结果如图 9.3 所示。

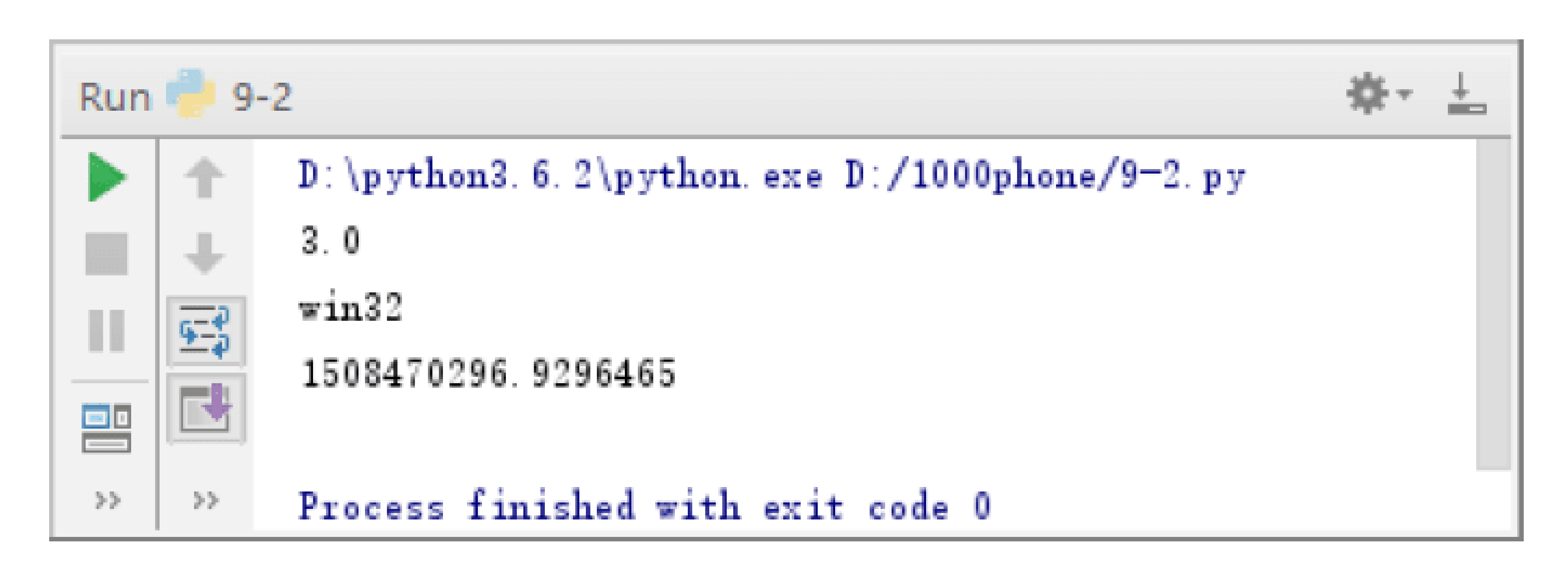


图 9.3 例 9-2 运行结果

在例 9-2 中，第 1 行为 math 模块指定别名 m，第 2 行一次导入多个模块。

此外，若只想导入模块中的某个对象，则可以使用 from 导入模块中的指定对象，其语法格式如下：

```
from 模块名 import 导入对象名            # 导入模块中某个对象
from 模块名 import 导入对象名 as 别名    # 给导入的对象指定别名
from 模块名 import *                     # 导入模块中所有对象
```

注意使用 from 导入的对象可以直接使用，不需要使用模块名作为限定符，如例 9-3 所示。

例 9-3 使用 from 导入对象。

```
1  from math import sqrt as st
2  print(st(9))
```

运行结果如图 9.4 所示。

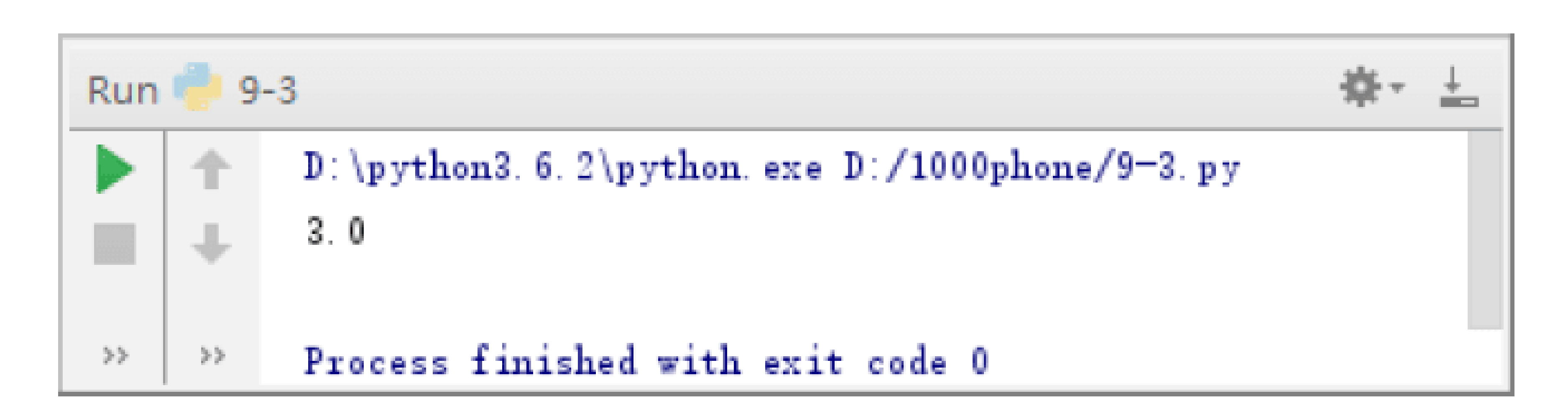


图 9.4 例 9-3 运行结果

在例 9-3 中，第 1 行通过 from 关键字从 math 模块中导入 sqrt()函数，然后再通过 as 关键字为 sqrt()函数指定别名 st。

import 与 from 导入模块各有特点，使用 import 导入模块时比较简单，使用 from 导入模块时需列出想要导入的对象名，但无论哪种导入方式，模块只能一次导入。另外，注意模块整体导入的开销是比较大的。

9.3　内置标准模块

Python 标准库中包括了许多模块，从 Python 语言自身特定的类型到一些只用于少数程序的模块，本节主要介绍基础阶段常见的内置标准模块。

9.3.1　sys 模块

sys 模块是 Python 标准库中最常用的模块之一。通过它可以获取命令行参数，从而实现从程序外部向程序内部传递参数的功能，也可以获取程序路径和当前系统平台等信息。

接下来演示通过 sys 模块获取命令行参数，如例 9-4 所示。

例 9-4　通过 sys 模块获取命令行参数。

```
import sys
print(sys.argv)
print("参数个数:" + str(len(sys.argv)))
for i in range(len(sys.argv)):
    print("" + str(i + 1) + ": " + sys.argv[i])
```

运行结果如图 9.5 所示。



图 9.5　例 9-4 运行结果

在例 9-4 中，注意执行程序时，需要开启终端模式(在 PyCharm 中，选择 View->Tool Windows->View->Terminal 选项即可)。从程序运行结果可以看出，在命令行中输入了 3

个参数，分别为'9-4.py'、'xiaoqian'、'666'。

在导入模块时，用户省略了模块文件的路径和扩展名，但 Python 解释器可以找到对应的文件，这是因为 Python 解释器会按特定的路径来搜索模块文件，用户可以通过 sys.path 获取搜索模块的路径，如例 9-5 所示。

例 9-5 通过 sys.path 获取搜索模块的路径。

```
1   import sys
2   print(sys.path)
```

运行结果如图 9.6 所示。

```
Run 9-5
D:\python3.6.2\python.exe D:/1000phone/9-5.py
['D:\\1000phone', 'D:\\1000phone', 'D:\\python3.6
 .2\\python36.zip', 'D:\\python3.6.2\\DLLs',
 'D:\\python3.6.2\\lib', 'D:\\python3.6.2',
 'D:\\python3.6.2\\lib\\site-packages']

Process finished with exit code 0
```

图 9.6 例 9-5 运行结果

在例 9-5 中，第 2 行通过 print()函数打印出搜索模块路径。sys.path 通常由 4 部分组成，具体如下所示：

- 程序的当前目录（可用 os 模块中的 getcwd()函数查看当前目录名称）。
- 操作系统的环境变量 PYTHONPATH 中包含的目录（如果存在）。
- Python 标准库目录。
- 任何.pth 文件包含的目录（如果存在）。

9.3.2 platform 模块

platform 模块提供了很多方法用于获取有关开发平台的信息，如例 9-6 所示。

例 9-6 platform 模块中的方法。

```
import platform
print(platform.platform())          # 获取当前操作系统名称及版本号
print(platform.architecture())      # 获取计算机类型信息
print(platform.python_build())      # 获取 Python 版本信息
print(platform.python_compiler())   # 获取 Python 编译器信息
```

运行结果如图 9.7 所示。

在例 9-6 中，通过 platform 模块可以获取有关开发平台的相关信息。

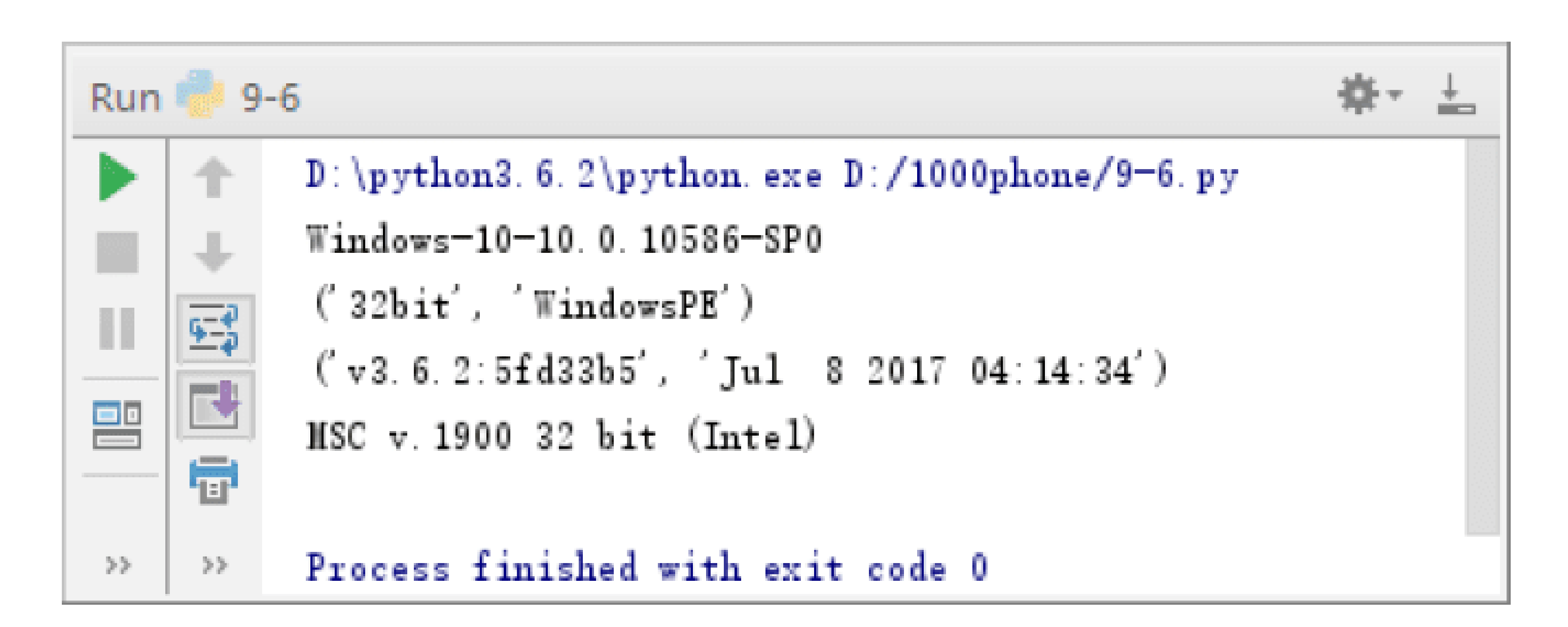


图 9.7　例 9-6 运行结果

9.3.3　random 模块

random 模块用于生成随机数，其中常用的函数如表 9.1 所示。

表 9.1　random 模块的主要函数

函　　数	说　　明
random()	返回一个 0 到 1 之间的随机浮点数 n(0≤n＜1)
uniform(a, b)	返回一个指定范围内的随机符点数 n(a≤n≤b 或 b≤n≤a)
randint(a, b)	返回一个指定范围内的整数 n(a≤n≤b)
randrange ([start], stop[, step])	从指定范围内按指定基数递增的集合中获取一个随机数
choice(sequence)	从序列中获取一个随机元素
shuffle (x[, random])	用于将一个列表中的元素打乱
sample(sequence, k)	从指定序列中随机获取指定长度 k 的片断，原有序列不会被修改

接下来演示 random 模块中主要函数的用法，如例 9-7 所示。

例 9-7　random 模块中主要函数的用法。

```
import random
print(random.random())                  # 生成 0～1 之间的一个随机浮点数
print(random.uniform(3, 5))             # 生成 3～5 之间的一个随机浮点数
print(random.uniform(5, 3))             # 生成 3～5 之间的一个随机浮点数
print(random.randint(0,5))              # 生成 0～5 之间的一个随机整数
print(random.randrange(0, 6, 2))        # 从 0、2、4 中随机获取一个数
list = [1, 2, 3, 4, 5, 6]
random.shuffle(list)                    # 打乱列表 list 中的元素
print(list)
print(random.sample(list, 4))           # 从列表 list 中随机获取 4 个元素
```

运行结果如图 9.8 所示。

在例 9-7 中，程序通过 random 模块可以生成随机数。注意每次运行程序时，结果可能会发生变化。

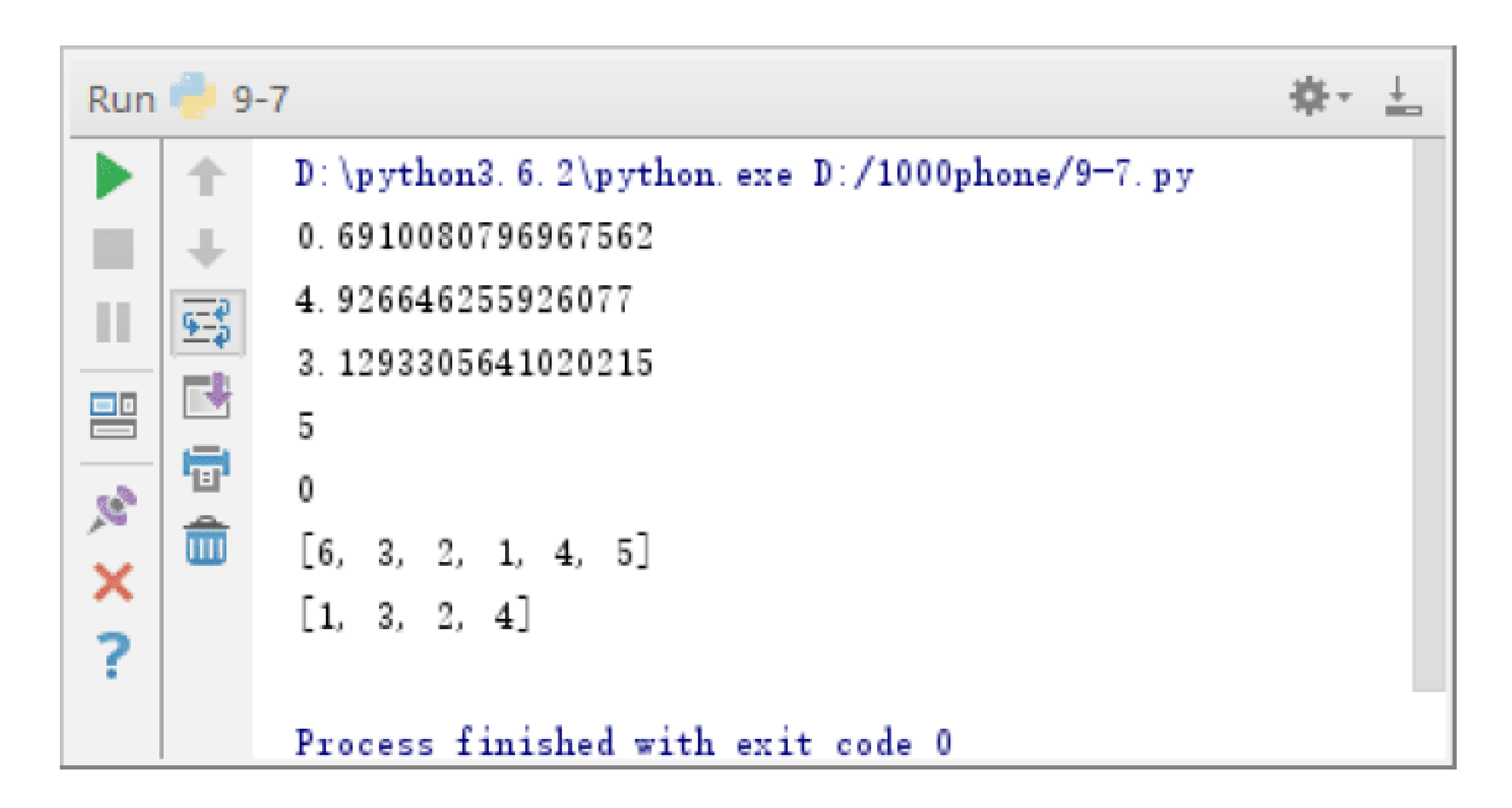


图 9.8 例 9-7 运行结果

9.3.4 time 模块

time 模块用于获取并处理时间，Python 中有两种时间表示方式，接下来分别介绍每种表示方式。

1．时间戳

时间戳是指从格林尼治时间 1970 年 01 月 01 日 00 时 00 分 00 秒（北京时间 1970 年 01 月 01 日 08 时 00 分 00 秒）起至现在的总秒数。time 模块中的 time()函数可以获取当前时间的时间戳，如例 9-8 所示。

例 9-8 time 模块中的 time()函数的用法。

```
import time
print(time.time())
```

运行结果如图 9.9 所示。

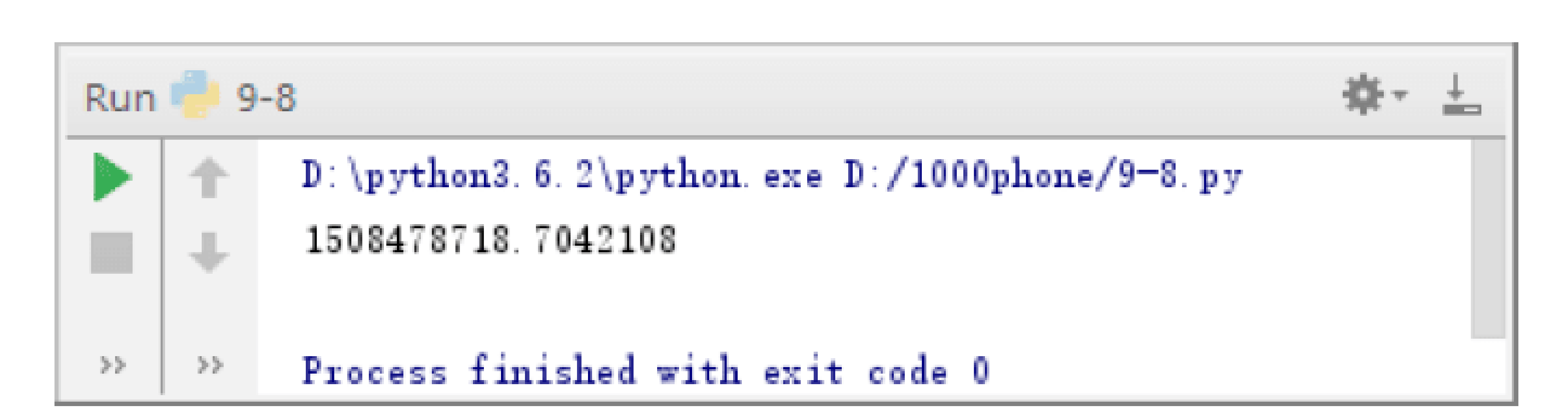


图 9.9 例 9-8 运行结果

在例 9-8 中，通过 time 模块中的 time()函数获取当前时间的时间戳，但从该结果中不能直接得出它所表示的时间，此时可以将该结果转换为时间元组，再进行格式化输出。

2．时间元组

时间元组 struct_time 包含 9 个元素，具体如表 9.2 所示。

表 9.2 struct_time 元组的字段

序 号	字 段	说 明
0	tm_year	年份，0000~9999 的整数
1	tm_mon	月份，1～12 的整数
2	tm_day	日期，1～31 的整数
3	tm_hour	小时，0～23 的整数
4	tm_min	分钟，0～59 的整数
5	tm_sec	秒钟，0～59 的整数
6	tm_wday	星期，0～6 的整数(星期一为 0)
7	tm_yday	天数，1～366 的整数
8	tm_isdst	0 表示标准时区，1 表示夏令时区

在 time 模块中，localtime()函数可以将一个时间戳转为一个当前时区的时间元组，如例 9-9 所示。

例 9-9 localtime()函数的用法。

```
import time
print(time.localtime(time.time()))
```

运行结果如图 9.10 所示。

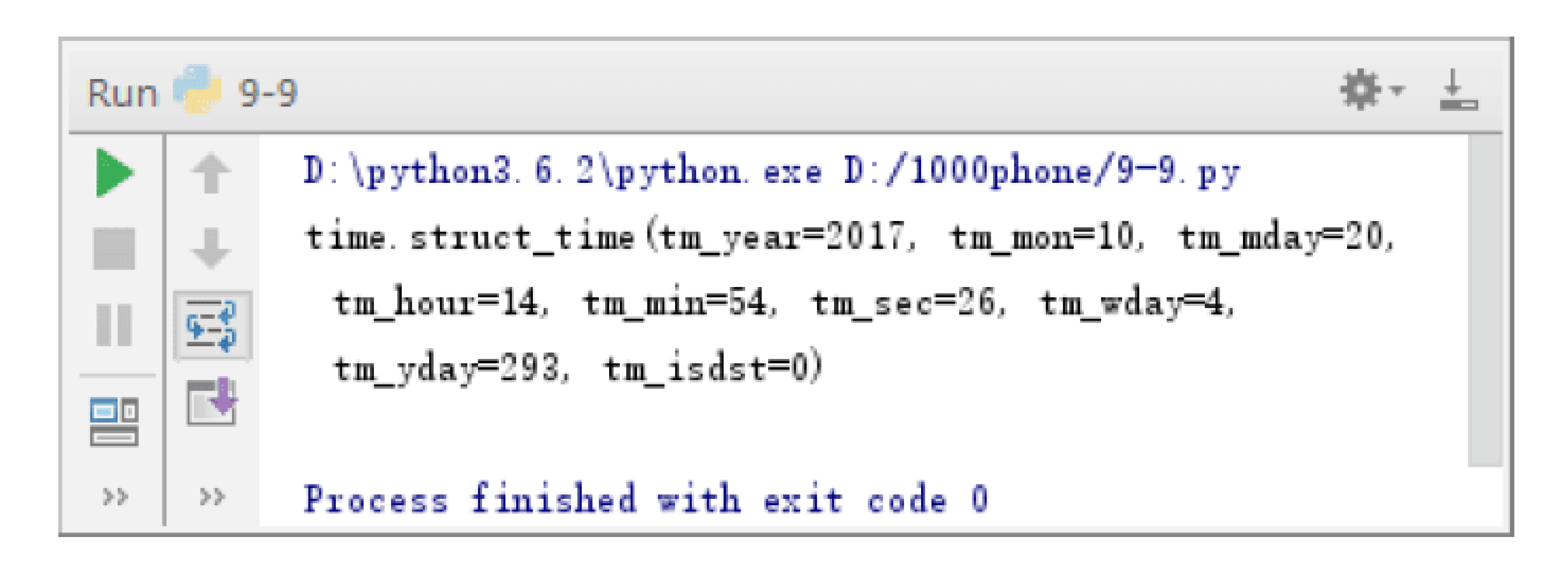


图 9.10 例 9-9 运行结果

在例 9-9 中，程序通过 time 模块中 localtime()函数可以将时间戳转为时间元组。从运行结果可发现，时间元组表示时间也不易观察，此时可以通过 strftime()函数将时间元组格式化，如例 9-10 所示。

例 9-10 strftime()函数的用法。

```
import time
print(time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())))
print(time.strftime('%a %b %d %H:%M:%S %Y', time.localtime(time.time())))
```

运行结果如图 9.11 所示。

在例 9-10 中，通过 time 模块中 strftime ()函数可以将时间元组格式化。该函数中第一个参数为格式化的时间字符串，其中格式化符号如表 9.3 所示。

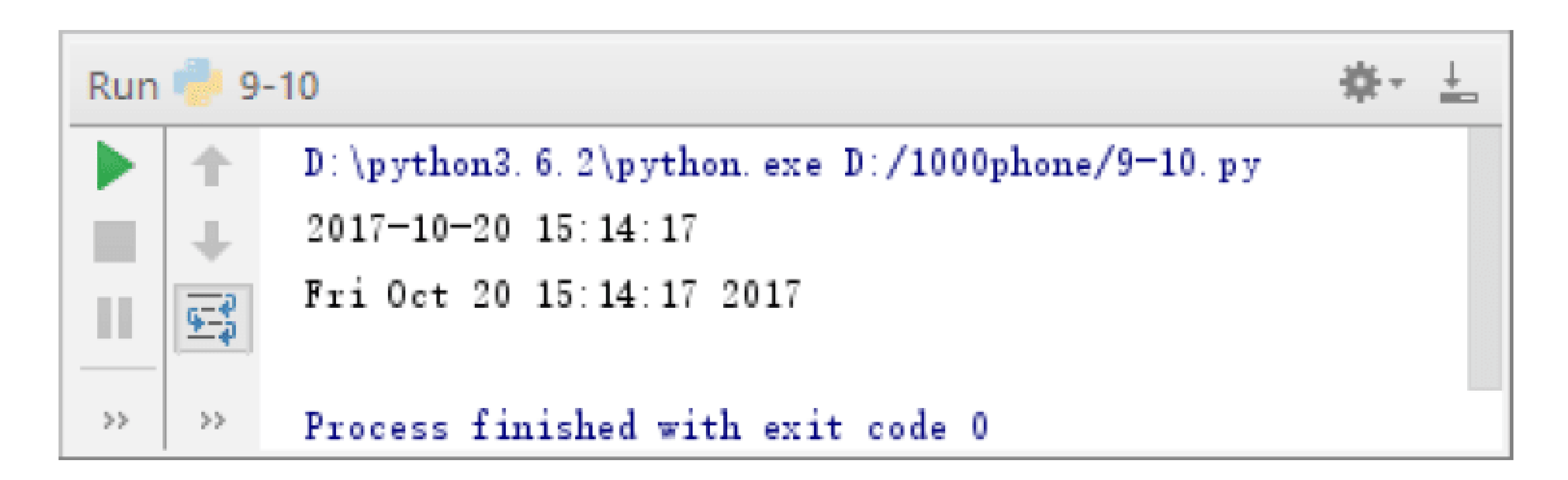


图 9.11 例 9-10 运行结果

表 9.3 时间格式化符号

格式化符号	说 明	格式化符号	说 明
%y	年份，00～99	%B	本地简化月份名称
%Y	年份，0000～9999	%c	本地相应的日期表示和时间表示
%m	月份，01~12	%j	天数，001～366
%d	天数，01~31	%p	本地 A.M.或 P.M.
%H	小时，00~23	%U	星期数，00～53，星期天为星期的开始
%I	小时，01~12	%w	星期几，0～6，星期天为星期的开始
%M	分钟，00~59	%W	星期数，00～53，星期一为星期的开始
%S	秒钟，00~59	%x	本地相应的日期表示
%a	本地简化星期名称	%X	本地相应的时间表示
%A	本地完整星期名称	%Z	当前时区的名称
%b	本地简化月份名称	%%	%号本身

Python 中还有许多内置标准模块，可以通过在终端模式下输入“help('模块名')”查看该模块包含的对象及用法，如通过 help('time')查看 time 模块的用法，如图 9.12 所示。

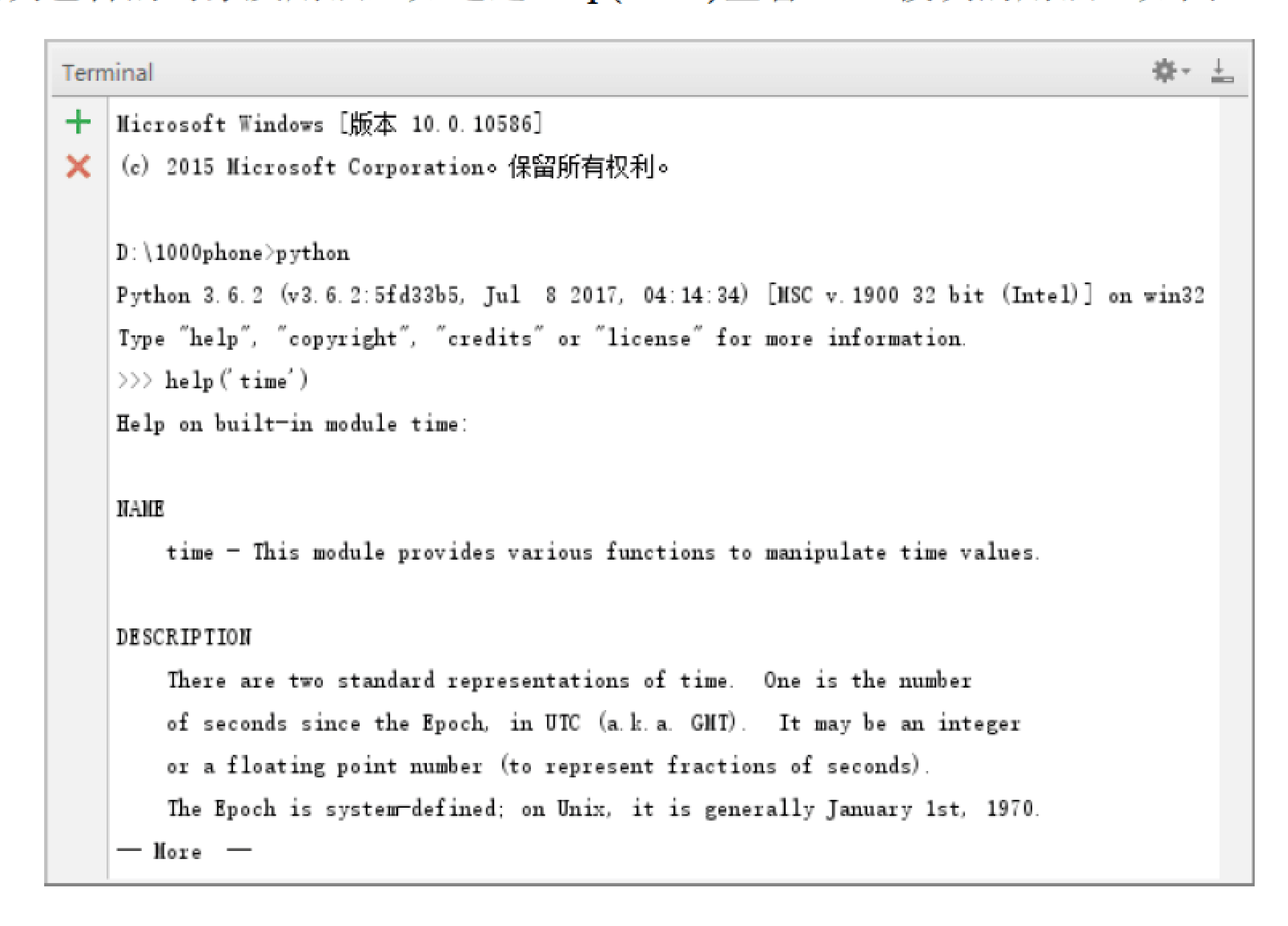


图 9.12 通过 help()查看模块用法

9.4 自定义模块

毕竟内置标准模块的功能有限，开发人员经常需要自定义函数，此时可以把函数组织到模块中，其他程序只需导入便可以引用模块中定义的函数，这种做法不仅使程序具有良好的结构，而且增加了代码的重用性。

在 Python 中，每个.py 文件都可以作为一个模块，模块的名字就是文件的名字，接下来演示如何自定义模块，假设 mymodule.py 文件中包含 2 个函数，具体如下所示：

```
# 输出信息
def output(info):
    print(info)
# 求和
def add(num1, num2):
    print(num1 + num2)
```

如果创建的模块 mymodule.py 与 9-11.py 保存在同一目录下，此时通过导入该模块便可引用其中包含的函数，如例 9-11 所示。

例 9-11 自定义模块。

```
import mymodule                                   # 导入 mymodule 模块
mymodule.output('千锋教育-中国 IT 职业教育领先品牌')  # 调用 output()函数
mymodule.add(1, 2)                                # 调用 add()函数
```

运行结果如图 9.13 所示。

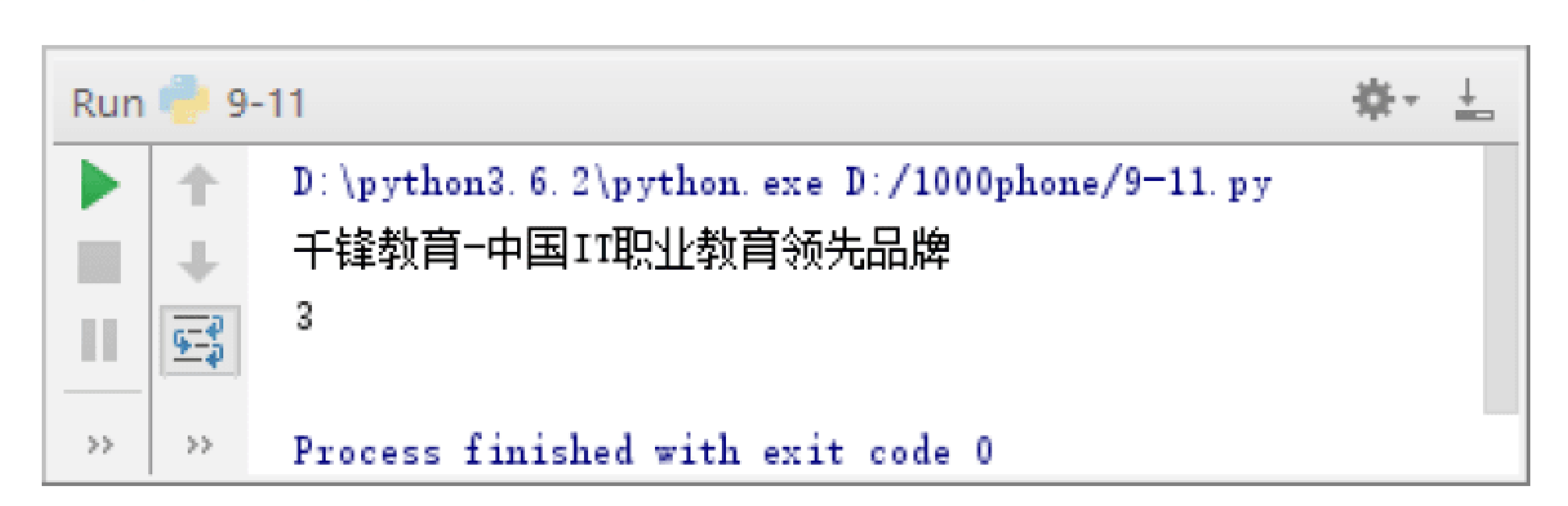


图 9.13 例 9-11 运行结果

在例 9-11 中，程序导入 mymodule 模块并引用其中包含的函数。

在实际开发中，自定义完模块后，为了保证模块编写正确，一般需要在模块中添加测试信息，具体如下所示：

```
# 输出信息
def output(info):
    print(info)
# 求和
```

```
def add(num1, num2):
    print(num1 + num2)
# 测试
output('扣丁学堂|好程序员特训营')
add(3, 5)
```

此时，若执行 9-11.py 代码，则运行结果如图 9.14 所示。

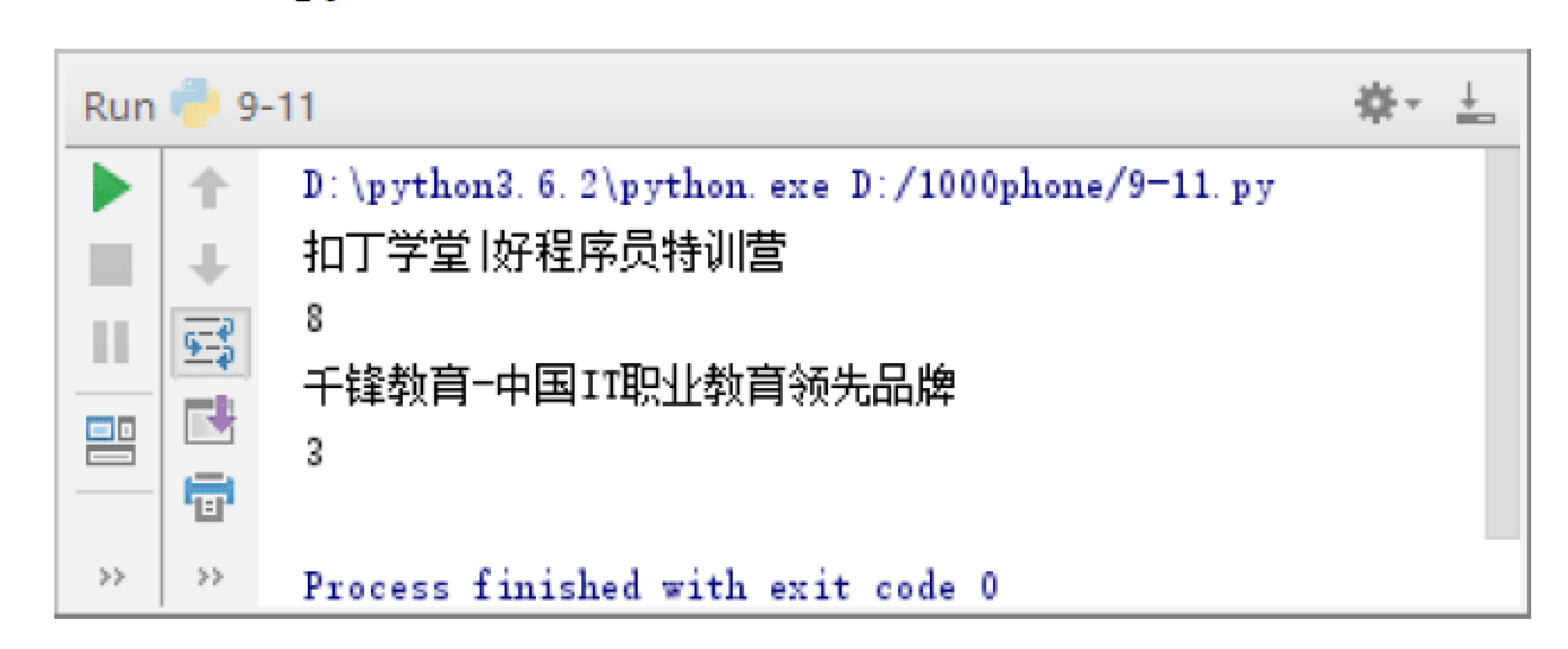


图 9.14　例 9-12 运行结果

从程序运行结果可发现，9-11.py 执行了 mymodule.py 中的测试代码，这是不期望出现的结果。为了解决上述问题，Python 提供了一个__name__属性，它存在于每个.py 文件中。当模块被其他程序导入使用时，模块__name__属性值为模块文件的主名；当模块直接被执行时，__name__属性值为'__main__'。

接下来修改 mymodule.py 文件，使其作为模块导入时不执行测试代码，具体如下所示：

```
# 输出信息
def output(info):
    print(info)
# 求和
def add(num1, num2):
    print(num1 + num2)
# 测试
if __name__ == '__main__':
    output('扣丁学堂|好程序员特训营')
    add(3, 5)
```

修改完成后，再次执行 9-11.py 代码，运行结果如图 9.15 所示。

```
Run 9-11
D:\python3.6.2\python.exe D:/1000phone/9-11.py
千锋教育-中国IT职业教育领先品牌
3

Process finished with exit code 0
```

图 9.15　例 9-13 运行结果

从程序运行结果可发现，执行 9-11.py 代码时，程序并没有执行 mymodule 模块中的测试代码。

9.5 包的概念

Python 的程序由包、模块和函数组成。包是由一系列模块组成的集合，模块是处理某一类问题的函数和类的集合，它们之间的关系如图 9.16 所示。

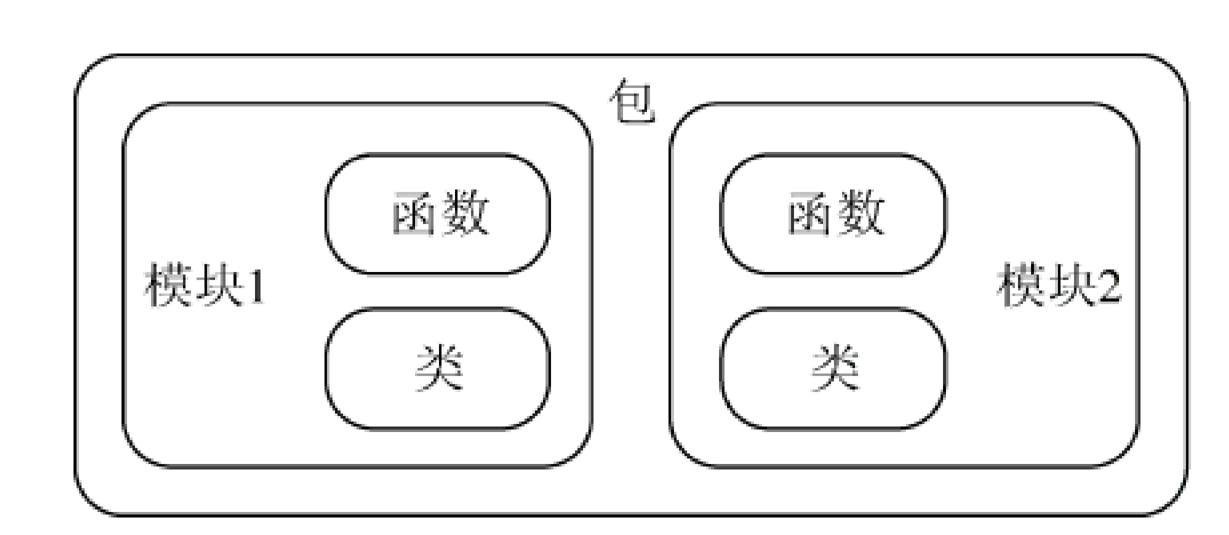


图 9.16　包与模块的关系

Python 提供了许多有用的工具包，如字符串处理、Web 应用、图像处理等，这些自带的工具包和模块安装在 Python 的安装目录下的 Lib 子目录中。包是一个至少包含__int__.py 文件的文件夹，__init__.py 文件一般用来进行包的某些初始化工作或者设置__all__值，其内容可以为空。

假设首先在包 pack 中创建两个子包：pack1 和 pack2，然后在包 pack1 中定义模块 myModule1，在包 pack2 中定义模块 myModule2，最后在包 pack 中定义一个模块 main，调用子包 pack1 和 pack2 中的模块，具体结构如下所示：

```
D:\1000PHONE\PACK
|   main.py
|   __init__.py
├─pack1
|       myModule1.py
|         __init__.py
└─pack2
        myModule2.py
          __init__.py
```

其中，pack1 包下的__init__.py 文件内容如下：

```
if __name__ == '__main__':
    print('作为主程序运行')
else:
    print('pack1 初始化')
```

pack1 包下的 myModule1.py 文件内容如下：

```
def func():
    print("调用 pack.pack1.myModule1.func()")
```

```
if   name   == '  main  ':
    print('作为主程序运行')
else:
    print('pack1 初始化')
```

pack2 包下的__init__.py 文件内容如下：

```
if   name   == '  main  ':
    print('作为主程序运行')
else:
    print('pack2 初始化')
```

pack2 包下的 myModule2.py 文件内容如下：

```
def func():
    print("调用 pack.pack2.myModule2.func()")
if   name   == '  main  ':
    print('作为主程序运行')
else:
    print('pack2 初始化')
```

pack 包下的__init__.py 文件内容如下：

```
if __name__ == '__main__':
    print('作为主程序运行')
else:
    print('pack 初始化')
```

pack 包下的 main.py 文件内容如下：

```
import pack.pack1.myModule1
from pack.pack2 import myModule2
pack.pack1.myModule1.func()
myModule2.func()
```

运行 main.py 程序，则运行结果如图 9.17 所示。

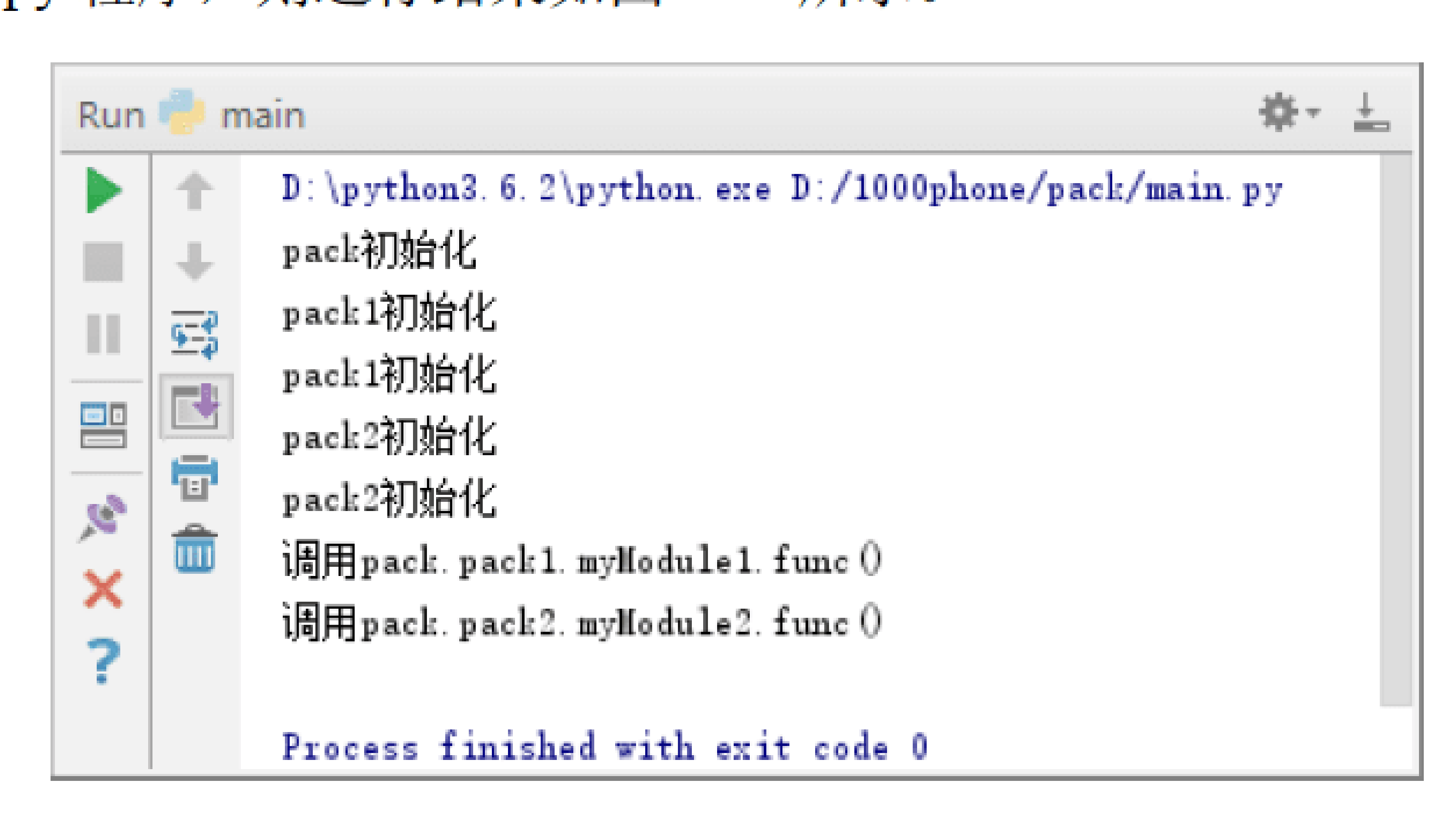


图 9.17　运行结果（一）

由程序运行结果可发现，导入模块过程中遇到的所有__init__.py 文件都会被执行。从包中导入单独的模块可以使用以下 3 种方法：

```
import Package.SubPackage.Module # 使用时必须用全路径名
from Package.SubPackage import Module # 可直接使用模块名而不用加包前缀
from Package.SubPackage.Module import function # 直接导入模块中的函数或变量
```

当需要导入某个包下的所有模块时，不可以直接使用如下语句：

```
from Package.SubPackage import *
```

这时需要使用__all__记录当前包所包含的模块，例如在 pack1 包的__init__.py 文件中第一行添加如下代码：

```
__all__ = ['myModule1']
```

其中中括号中的内容是模块名的列表，如果模块数量超过两个，使用逗号分开。同理，在 pack2 包的__init__.py 文件中也添加一行类似的代码：

```
__all__ = ['myModule2']
```

这样就可以在 main.py 模块中一次导入 pack1、pack2 包中所有的模块，pack 包下的 main.py 文件内容如下：

```
from pack.pack1 import *
from pack.pack2 import *
myModule1.func()
myModule2.func()
```

运行 main.py 程序，运行结果如图 9.18 所示。

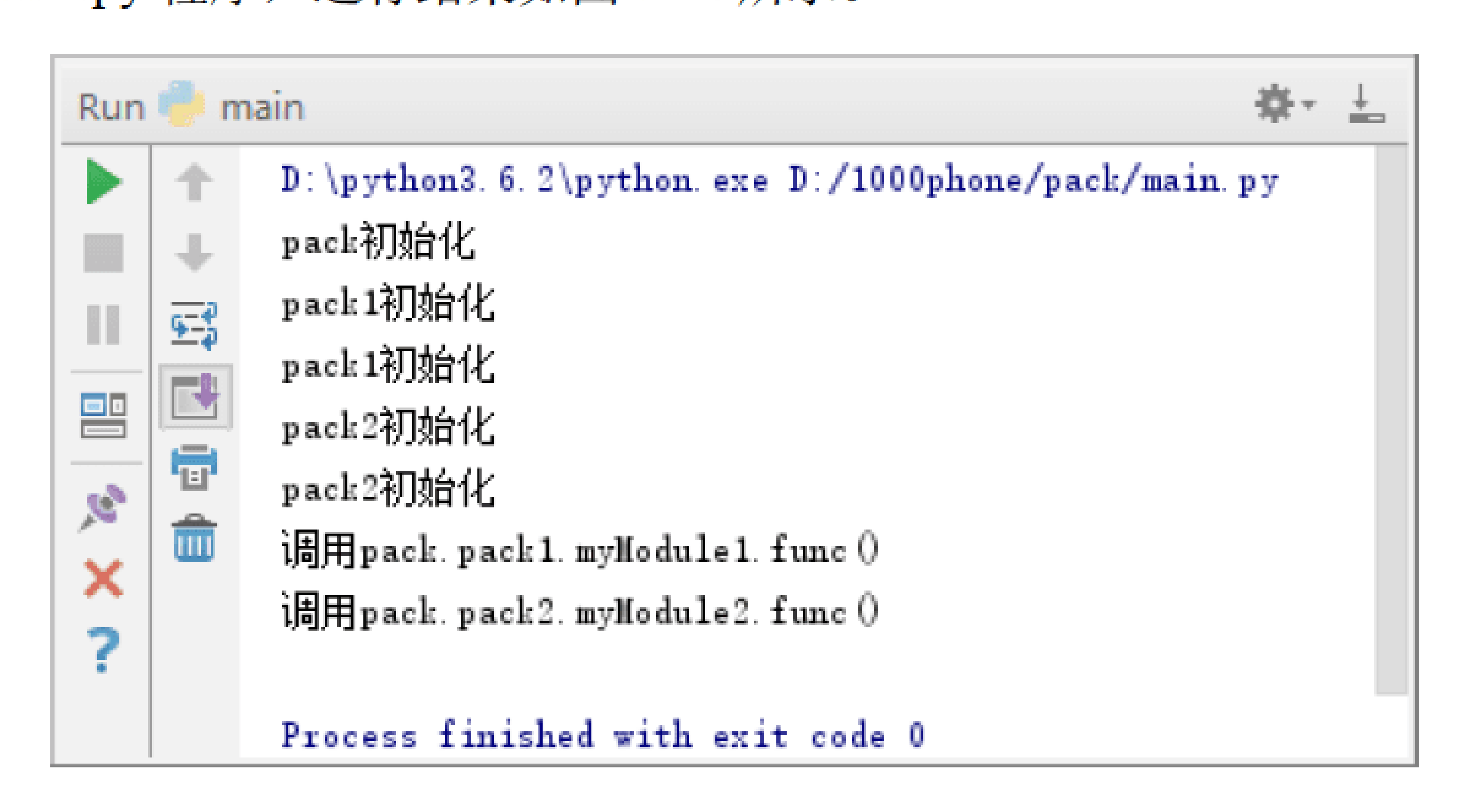


图 9.18 运行结果（二）

如果 pack1 包下的 myModule1.py 文件需要引用 pack2 包下的 myModule2 模块，默认情况下，myModule1.py 文件是找不到 myModule2 模块，此时需要按相对位置引入模块，修改 myModule1.py 文件，具体如下所示：

```
from .. import pack2
def func():
    pack2.myModule2.func()
    print("调用 pack.pack1.myModule1.func()")
if __name__ == '__main__':
    print('作为主程序运行')
else:
    print('pack1 初始化')
```

其中，“..”表示包含 from 导入命令的模块文件所在路径的上一级目录，运行 main.py 程序，则运行结果如图 9.19 所示。

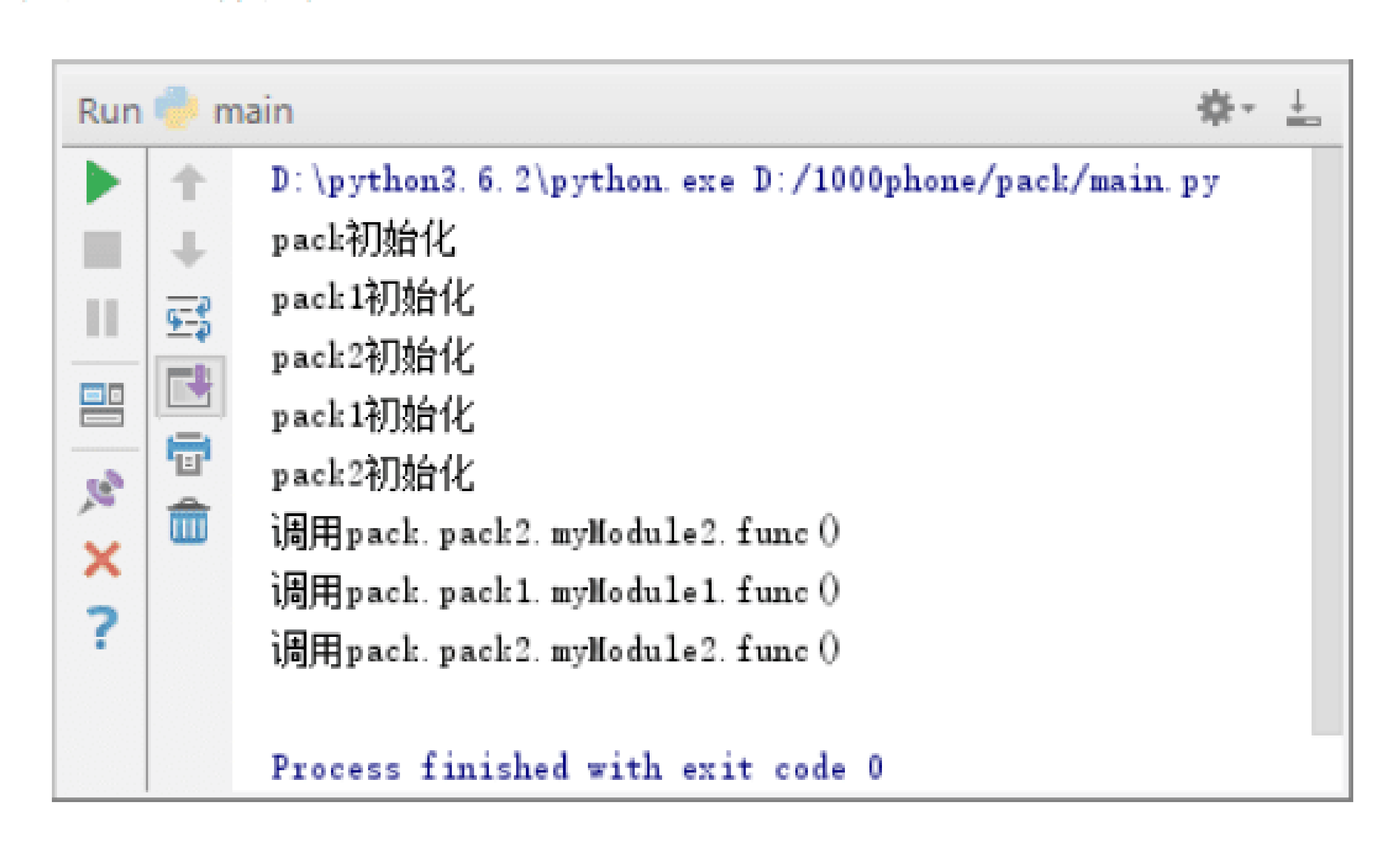


图 9.19 运行结果（三）

9.6 包的发布

本节将演示如何将写好的模块进行打包和发布，最简单的方法是将包直接复制到 Python 的 lib 目录下，但此方式不便于管理与维护，存在多个 Python 版本时会非常混乱。接下来通过编写 setup.py 来对 9.5 节介绍的 pack 模块进行打包。

（1）在 pack 所在的文件目录下新建 setup.py、MANIFEST.in、README.txt 文件，其目录结构如下：

```
D:\1000phone
|  MANIFEST.in
|  README.txt
|  setup.py
└─pack
    |  main.py
    |    init  .py
    ├─pack1
    |      myModule1.py
    |      __init__.py
```

```
    └─pack2
            myModule2.py
            __init__.py
```

（2）打开 setup.py 文件，编辑其内容如下：

```
from distutils.core import setup
setup(
    name = 'myProject',
    author = '千锋教育',
    author email = 'xiaoqian@1000phone.com',
    version = '1.0',
    url = 'http://www.qfedu.com/',
    packages = ['pack', 'pack.pack1', 'pack.pack2']
)
```

其中，packages 指明将要发布的包。

打开 MANIFEST.in 文件，编辑其内容如下：

```
include *.txt
```

该文件列出了各种希望包含在包中的非源代码。

README.txt 文件中的内容为提示使用者如何使用该包中的模块。

（3）在终端模式下，进入 pack 包所在的文件目录执行如下命令：

```
python setup.py build
```

该命令可以构建包，具体执行过程如图 9.20 所示。



图 9.20　构建包

执行完该命令后，目录结构如下所示：

```
D:\1000phone
```

```
|   MANIFEST.in
|   README.txt
|   setup.py
├─build
|   └─lib
|       └─pack
|           |   main.py
|           |   __init__.py
|           ├─pack1
|           |       myModule1.py
|           |       __init__.py
|           └─pack2
|                   myModule2.py
|                   __init__.py
└─pack
    |   main.py
    |   __init__.py
    ├─pack1
    |       myModule1.py
    |       __init__.py
    └─pack2
            myModule2.py
            __init__.py
```

（4）接着在终端模式下输入以下命令：

```
python setup.py sdist
```

该命令可以生成最终发布的压缩包，具体执行过程如图 9.21 所示。

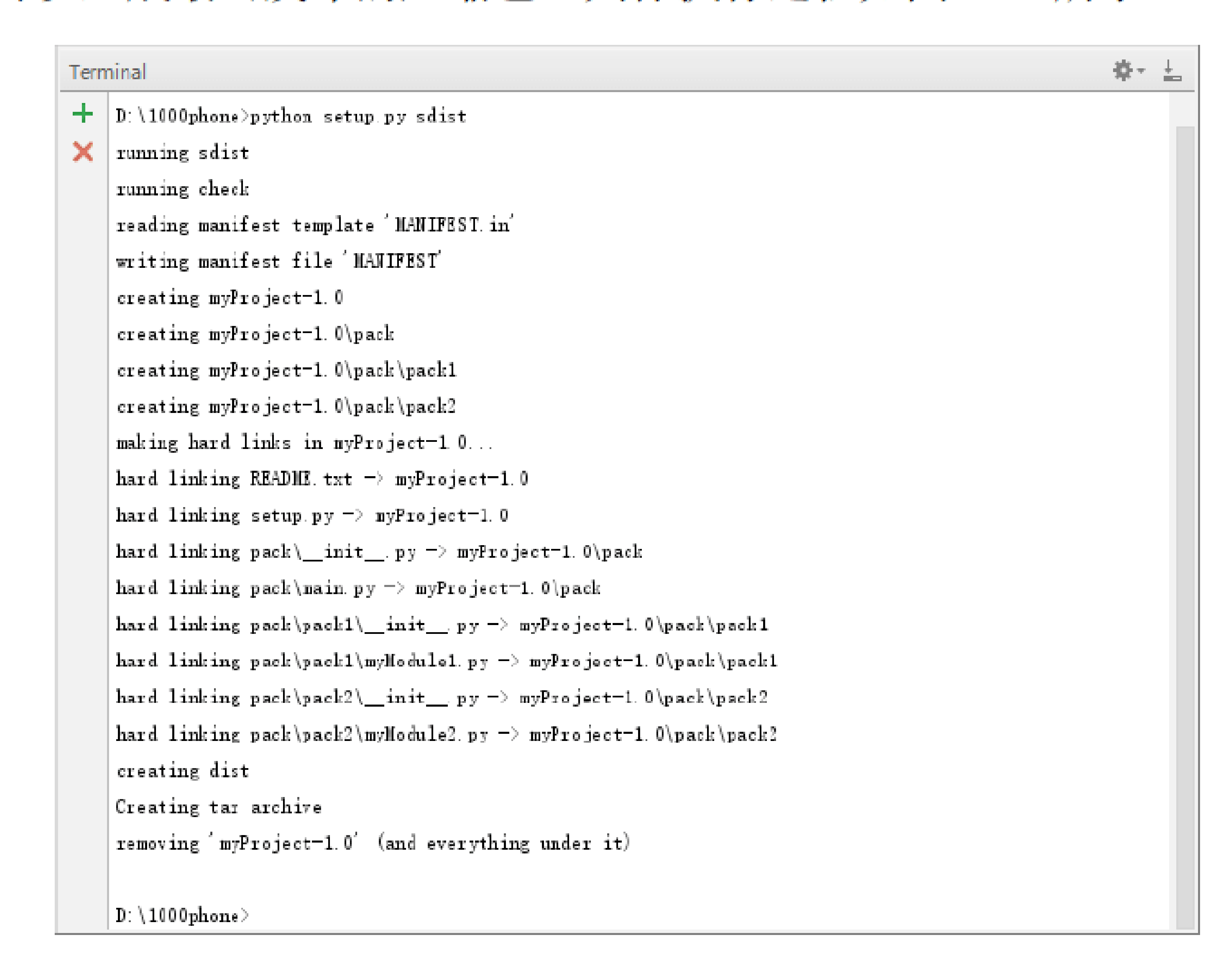


图 9.21　生成压缩包

执行完该命令后，目录结构如下所示：

```
D:\1000phone
│  MANIFEST
│  MANIFEST.in
│  README.txt
│  setup.py
├─build
│  └─lib
│      └─pack
│          │  main.py
│          │  __init__.py
│          ├─pack1
│          │      myModule1.py
│          │      __init__.py
│          └─pack2
│                  myModule2.py
│                  __init__.py
├─dist
│      myProject-1.0.tar.gz
└─pack
    │  main.py
    │  __init__.py
    ├─pack1
    │      myModule1.py
    │      __init__.py
    └─pack2
            myModule2.py
            __init__.py
```

其中，dist 目录下 myProject-1.0.tar.gz 文件为将要发布的包，开发者可以将此文件发布给其他人或上传到 Python 社区供更多的开发者下载。

9.7　包 的 安 装

9.6 节讲解了如何发布自己制作的包，本节讲解如何安装其他开发者发布的包（以 9.6 节最终生成的压缩包为例）。

（1）进入压缩包所在的文件目录并对其进行解压，解压后的文件目录如下所示：

```
D:\1000phone\dist
```

```
|   myProject-1.0.tar.gz
└─myProject-1.0
    |   PKG-INFO
    |   README.txt
    |   setup.py
    └─pack
        |   main.py
        |   __init__.py
        ├─pack1
        |       myModule1.py
        |       __init__.py
        └─pack2
                myModule2.py
                __init__.py
```

（2）在终端模式下，进入 myProject-1.0 目录下执行如下命令：

```
python setup.py install
```

该命令的具体执行过程如图 9.22 所示。

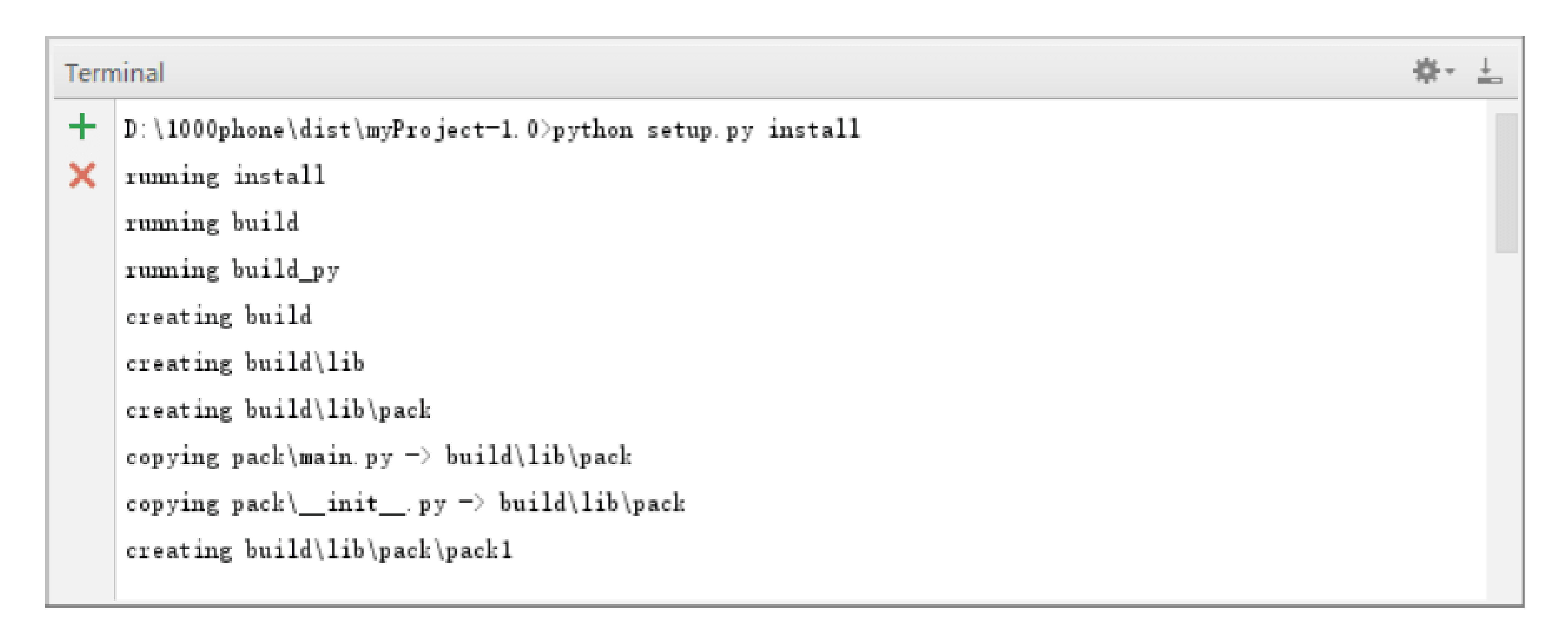


图 9.22　包安装过程

通过该命令就可以将 pack 包安装到系统（即 Python 路径）中，即该包存在于 D:\python3.6.2\Lib\site-packages（本书中 Python 的安装目录为 D:\python3.6.2）。

9.8　小　案　例

大家在使用手机 QQ 进行群聊时，经常会有小伙伴发红包，如图 9.23 所示。

发红包模块需要满足以下 3 点要求：

- 设定红包金额与数量。



图 9.23　发红包界面

- 金额数最大的为运气王。
- 随机分配金额。

接下来按照上述要求编写发红包功能模块，具体实现如例 9-12 所示。

例 9-12　发红包功能模块。

```
import random
def giveRedPackets(total, num):
    # total 表示拟发红包总金额
    # num 表示拟发红包数量
    print('共',total,'元  分', num, '份')
    each = []
    already = 0 # 已发红包总金额
    average = total / num # 平均金额
    for i in range(1, num):
        # 为当前抢红包的人随机分配金额
        # 至少给剩下的人每人留平均金额
        t1 = random.uniform(0, (total-already)-(num-i)*average)
        t = round(t1, 2)
        each.append(t)
        already += t
    # 剩余所有钱发给最后一个人
    each.append(round(total-already, 2))
    print("运气王:", sorted(each)[num - 1])
    random.shuffle(each)
```

```
        print('红包序列:', each)
        return each
if __name__=='__main__':
        total, num = 100, 6
        each = giveRedPackets(total, num)
```

运行结果如图 9.24 所示。

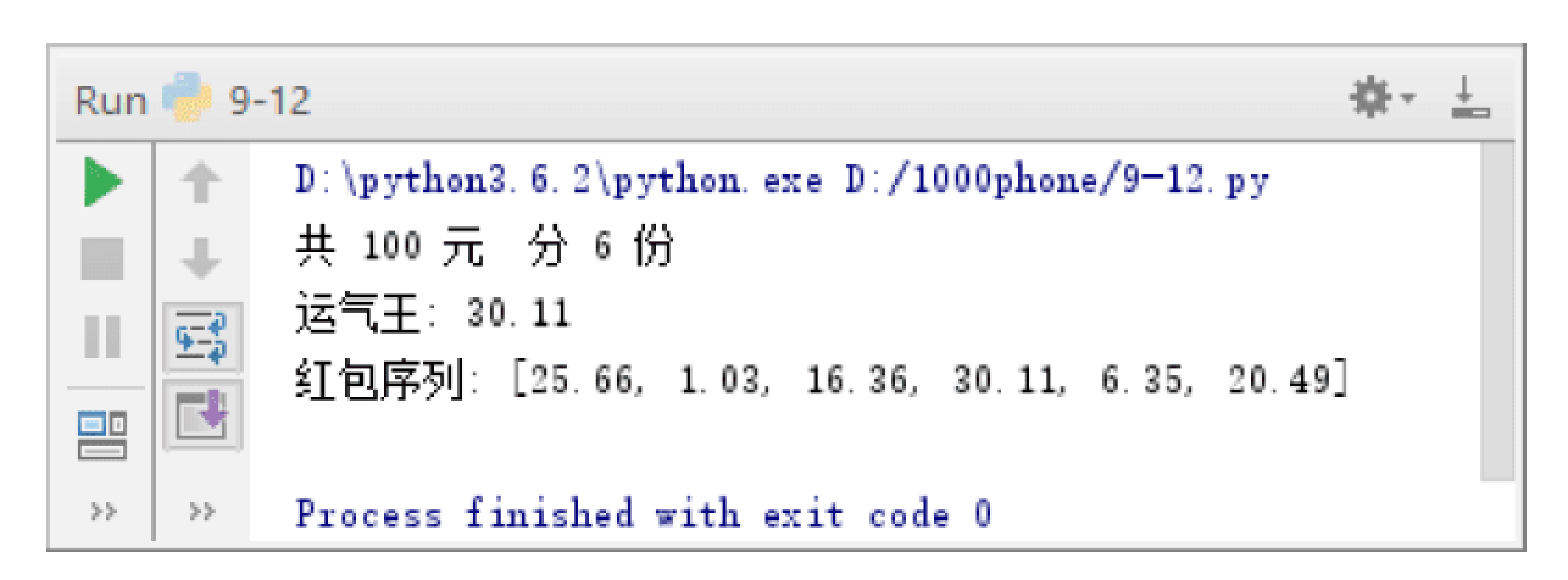


图 9.24 例 9-12 运行结果

在例 9-12 中，由于发红包时时，每个人所得的金额是随机的，因此需导入 random 模块。另外，round()函数用于返回浮点数的四舍五入值（对于 Python 2.x 版本是常规四舍五入，对于 Python 3.x 版本是奇进偶不进式的四舍五入），第一个参数表示需要进行四舍五入的浮点数，第二个参数表示从小数点到最后四舍五入的位数。

在 Python 中不建议使用浮点数进行精确计算，因为往往会得到意想不到的结果。大家可在实际开发中导入 decimal 模块来处理浮点数存在的误差。

9.9 本章小结

本章主要介绍了 Python 程序中的模块与包，包括模块的概念、模块的导入、内置标准模块、自定义模块、包的概念、包的发布及安装。在开发应用时可以将程序组织成模块架构，这样不仅可以提高代码的重用性，而且便于将复杂任务分解并进行分块调试。

9.10 习题

1．填空题

（1）在 Python 中，使用________关键字引入模块。
（2）导入模块时可以使用________关键字为模块指定别名。
（3）若导入模块中的某个对象，则可以使用________关键字。
（4）在安装包时，需执行________命令安装。
（5）在生成发布压缩包时，需执行________命令压缩。

2．选择题

（1）下列导入模块方式中，错误的是（　　）。

A．import math　　B．import math as m

C．from * import math　　D．from math import *

（2）每个模块内部都有一个（　　）属性。

A．__name__　　B．__main__

C．name　　D．'__main__'

（3）（　　）模块可以获取命令行参数。

A．platform　　B．random

C．sys　　D．time

（4）包目录下必须有一个（　　）文件。

A．__name__.py　　B．__init__.py

C．__main__.py　　D．__package__.py

（5）在 random 模块中，用于生成一个[0,1)之间的随机浮点数的函数是（　　）。

A．random(0, 1)　　B．randrange(0, 1)

C．random()　　D．randint(0, 1)

3．思考题

（1）使用模块有哪些优势？

（2）模块有哪些类型？

4．编程题



编写程序实现猜数字游戏模块，要求系统随机产生一个整数，玩家最多可以猜 5 次，系统会根据玩家的猜测进行提示，玩家则可以根据系统的提示对下一次的猜测进行适当调整。

第 10 章 chapter 10

面向对象（上）

本章学习目标

- 理解对象与类的概念。
- 掌握类的定义与对象的创建。
- 掌握构造方法与析构方法。
- 掌握类方法与静态方法。
- 掌握运算符重载。

面向对象程序设计是模拟如何组成现实世界而产生的一种编程方法，是对事物的功能抽象与数据抽象，并将解决问题的过程看成一个分类演绎的过程。其中，对象与类是面向对象程序设计的基本概念。

10.1 对象与类

在现实世界中，随处可见的一种事物就是对象，对象是事物存在的实体，如学生、汽车等。人类解决问题的方式总是将复杂的事物简单化，于是就会思考这些对象都是由哪些部分组成的。通常都会将对象划分为两个部分，即静态部分与动态部分。顾名思义，静态部分就是不能动的部分，这个部分被称为“属性”，任何对象都会具备其自身属性，如一个人，其属性包括高矮、胖瘦、年龄、性别等。然而具有这些属性的人会执行哪些动作也是一个值得探讨的部分，这个人可以转身、微笑、说话、奔跑，这些是这个人具备的行为（动态部分），人类通过探讨对象的属性和观察对象的行为来了解对象。

在计算机世界中，面向对象程序设计的思想要以对象来思考问题，首先要将现实世界的实体抽象为对象，然后考虑这个对象具备的属性和行为。例如，现在面临一名足球运动员想要将球射进对方球门这个实际问题，试着以面向对象的思想来解决它。步骤如下：

首先可以从这一问题中抽象出对象，这里抽象出的对象为一名足球运动员。

然后识别这个对象的属性。对象具备的属性都是静态属性，如足球运动员有一个鼻子、两条腿等，这些属性如图 10.1 所示。

接着识别这个对象的动态行为，即足球运动员的动作，如跳跃、转身等，这些行为都是这个对象基于其属性而具有的动作，这些行为如图 10.2 所示。

识别出这个对象的属性和行为后，这个对象就被定义完成了，然后根据足球运动员

具有的特性制定要射进对方球门的具体方案以解决问题。

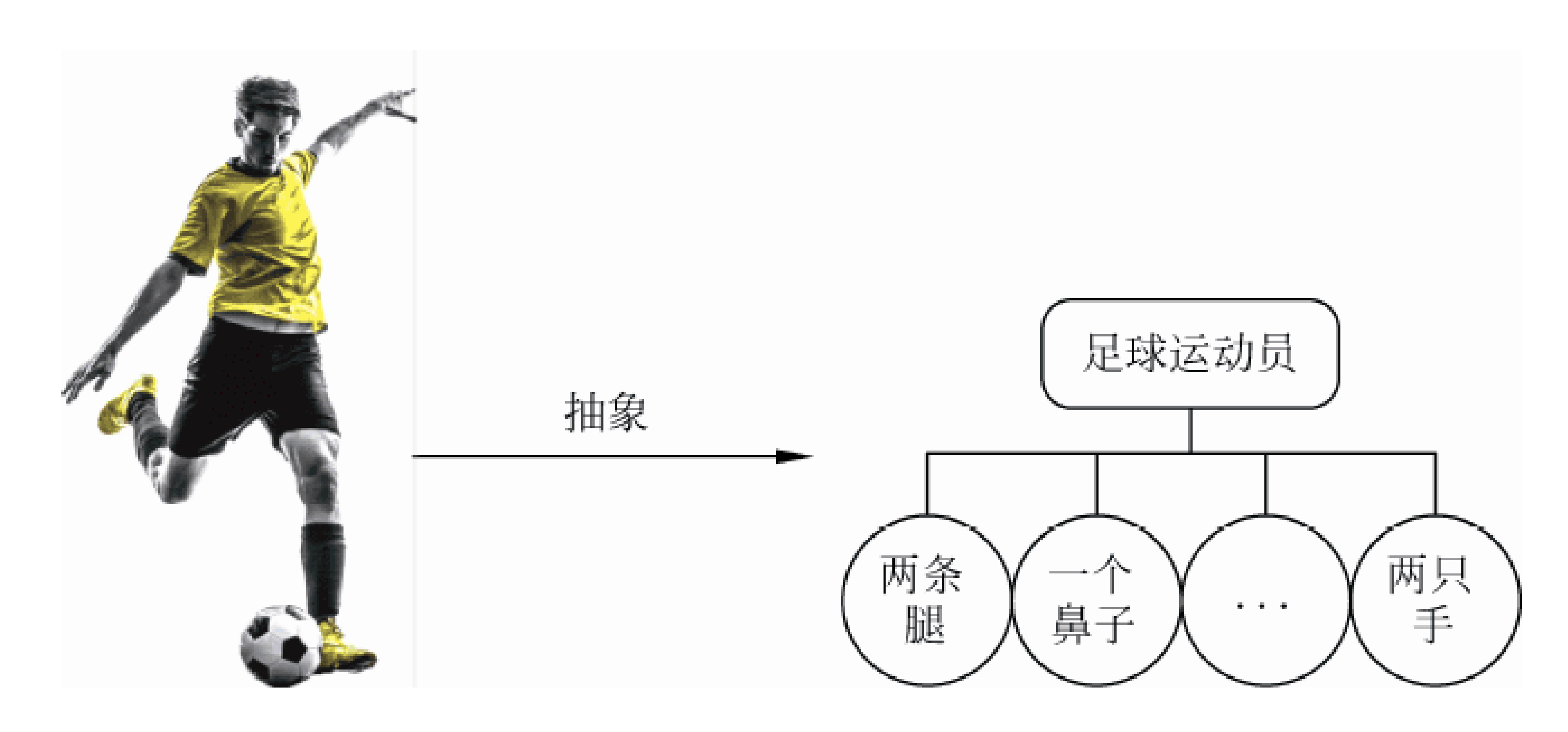


图 10.1 识别对象的属性

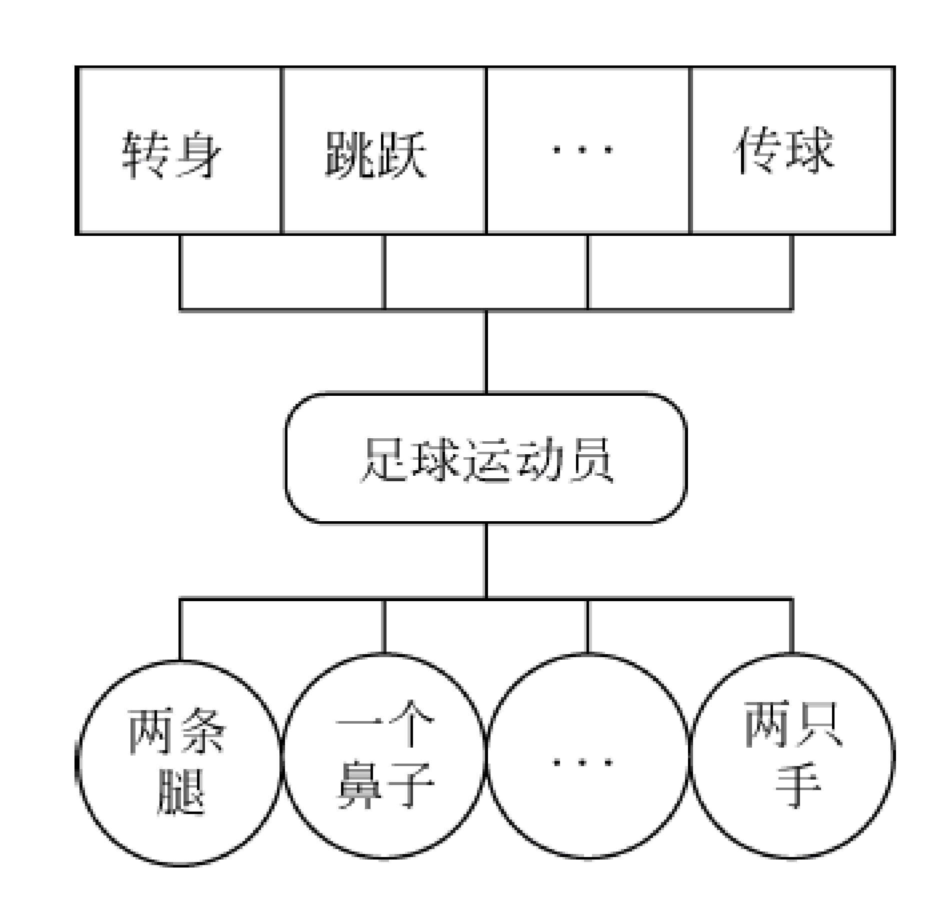


图 10.2 识别对象具有的行为

究其本质，所有的足球运动员都具有以上的属性和行为，可以将这些属性和行为封装起来以描述足球运动员这类人。由此可见，类实质上就是封装对象属性和行为的载体，而对象则是类抽象出来的一个实例。这也是进行面向对象程序设计的核心思想，即把具体事物的共同特征抽象成实体概念，有了这些抽象出来的实体概念，就可以在编程语言的支持下创建类。因此，类是那些实体的一种模型，具体如图 10.3 所示。

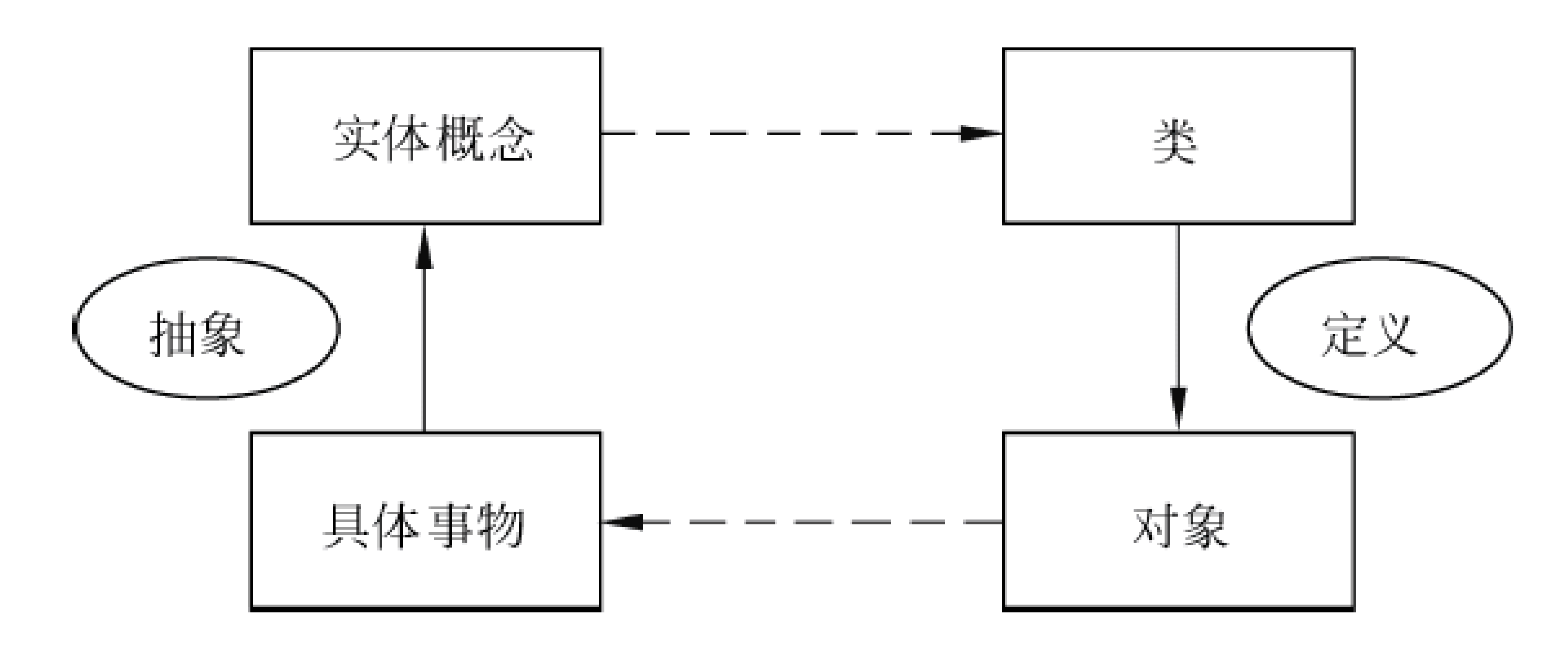


图 10.3 现实世界与编程语言的对应关系

在图 10.3 中，通过面向对象程序设计的思想可以建立现实世界中具体事物、实体概念与编程语言中类、对象之间的一一对应关系。

10.2 类的定义

Python 使用 class 关键字来定义类，其语法格式如下：

```
class 类名:
    类体
```

其中，类名的首字母一般需要大写，具体示例如下：

```
class Student:
    def say(self, name):     # 实例方法
        self.name = name     # 实例属性
        print('我是', self.name)
```

其中，实例方法与前面学习的函数格式类似，区别在于类的所有实例方法都必须至少有一个名为 self 的参数，并且必须是方法的第一个形参（如果有多个形参），self 参数代表将来要创建的对象本身。另外，self.name 称为实例属性，在类的实例方法中访问实例属性时需要以 self 为前缀。

在类中定义实例方法时，第一个参数指定为 self 只是一个习惯。实际上，该参数的名字是可以变化的，具体如下所示：

```
class Student:
    def say(my, name):       # 实例方法
        my.name = name       # 实例属性
        print('我是', my.name)
```

尽管如此，本书建议大家编写代码时仍以 self 作为实例方法的第一个参数名字，这样便于其他人阅读代码。

10.3 对象的创建

在 Python 中，有两种对象：类对象与实例对象。类对象只有一个，而实例对象可以有多个。

10.3.1 类对象

类对象是在执行 class 语句时创建的，如例 10-1 所示。

例 10-1 类对象。

```
class Student:
```

```
    def say(self, name):      # 实例方法
        self.name = name      # 实例属性
        print('我是', self.name)
    print('类对象生成')
print(type(Student))
```

运行结果如图 10.4 所示。

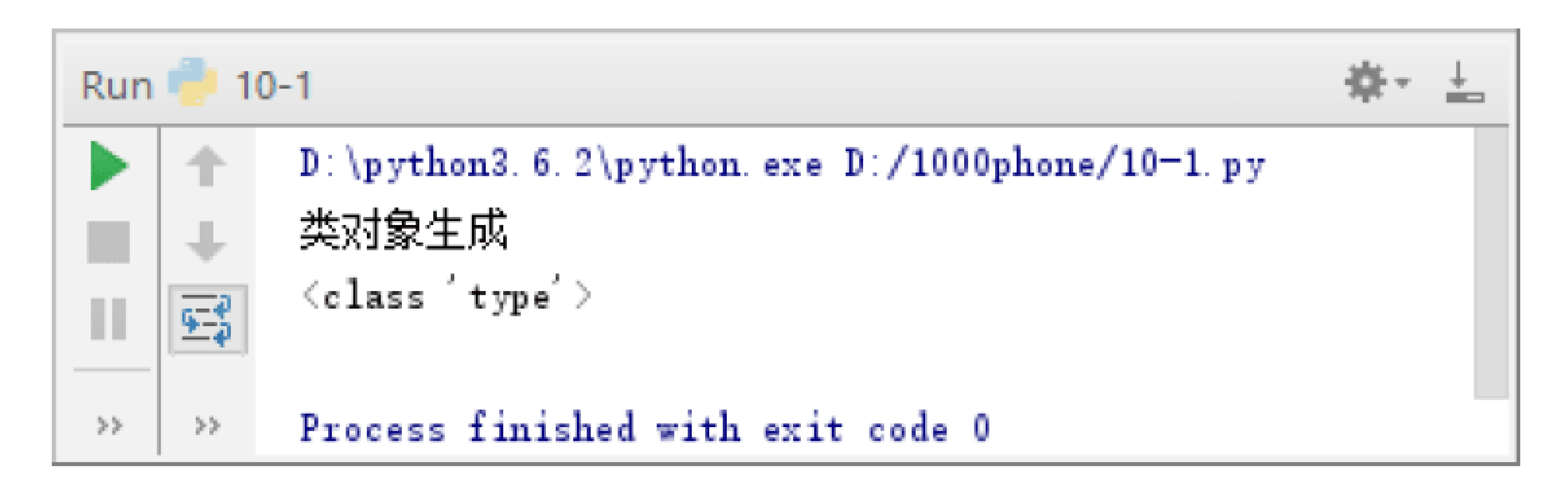


图 10.4 例 10-1 运行结果

在例 10-1 中，Python 执行 class 语句时创建了一个类对象和一个变量（名称就是类名称），变量引用类对象。通过 type()函数可以测试对象的类型。

在定义类时，还可以定义类属性，如例 10-2 所示。

例 10-2 定义类属性。

```
1   class Student:
2       school = '扣丁学堂'        # 类属性
3       def say(self, name):      # 实例方法
4           self.name = name      # 实例属性
5           print('我是', self.name)
6   print(Student.school)
```

运行结果如图 10.5 所示。

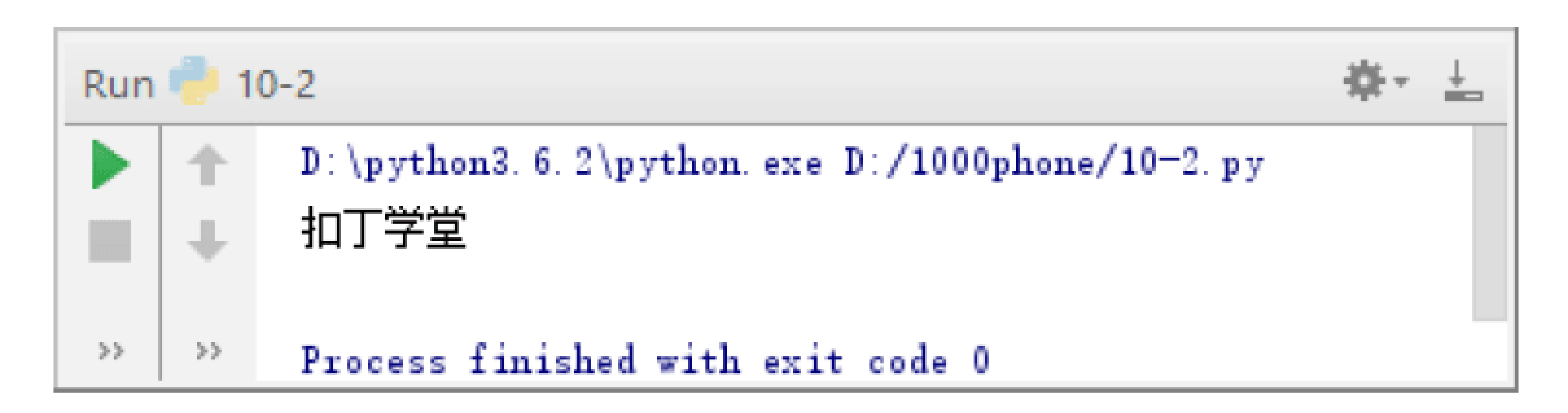


图 10.5 例 10-2 运行结果

在例 10-2 中，第 6 行通过“类名.类属性名”方式访问类属性。

10.3.2 实例对象

实例对象通过调用类对象来创建（就像调用函数一样来调用类对象），每个实例对

象继承类对象的属性，并获得自己的命名空间。实例方法的第一个参数默认为 self，表示引用实例对象。在实例方法中对 self 的属性赋值才会创建属于实例对象的属性，如例 10-3 所示。

例 10-3 创建实例对象属性。

```
1  class Student:
2      school = '扣丁学堂'            # 类属性
3      def say(self, name):          # 实例方法
4          self.name = name          # 实例属性
5          print('我是', self.name)
6  s1 = Student()                    # 创建实例对象
7  s1.say('小千')
8  print(s1.school, s1.name)
```

运行结果如图 10.6 所示。

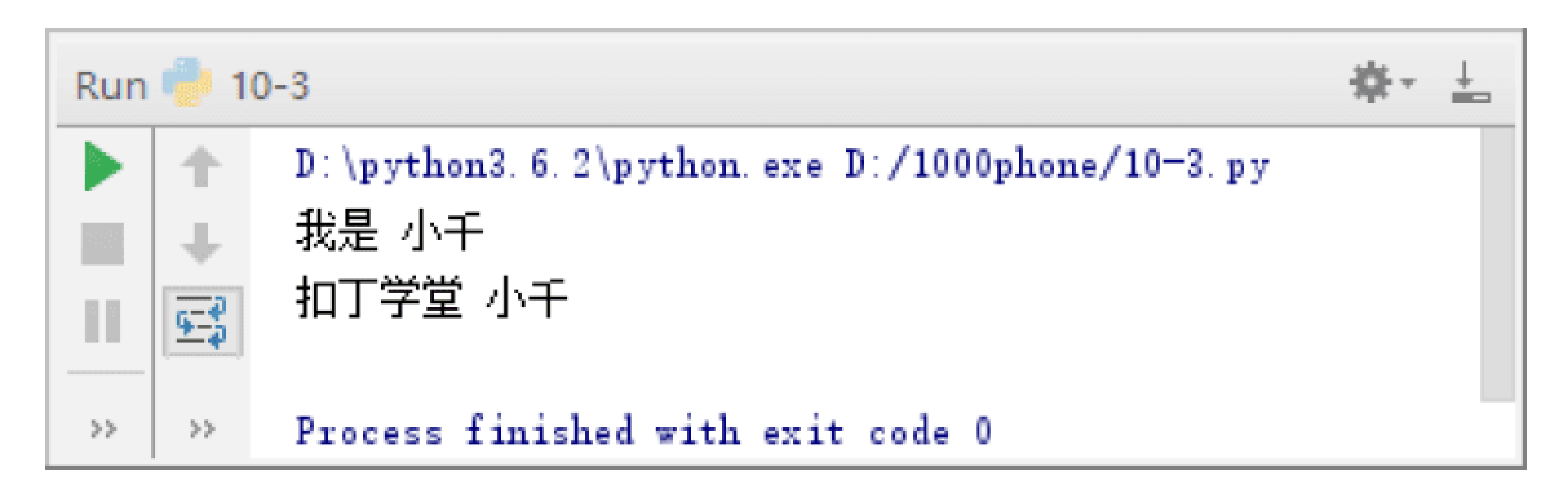


图 10.6 例 10-3 运行结果

在例 10-3 中，第 7 行通过“实例名.实例方法”方式调用实例方法，第 8 行通过实例对象访问类属性 school。

如果类中存在相同名称的类属性与实例属性，则通过实例对象只能访问实例属性，如例 10-4 所示。

例 10-4 通过实例对象只能访问实例属性。

```
class Student:
    school = '扣丁学堂'            # 类属性
    def say(self, school):        # 实例方法
        self.school = school      # 实例属性
        print('学校:', self.school)
s1 = Student()                    # 创建实例对象
s1.say('好程序员特训营')
print(Student.school, s1.school)
```

运行结果如图 10.7 所示。

在例 10-4 中，类属性名与实例属性名相同，通过实例对象访问属性时会获取实例属性名对应的值。因此，当程序中访问类属性时，一般通过类对象获取。

此外，还可以通过赋值运算符修改或增加类对象与实例对象的属性，如例 10-5 所示。

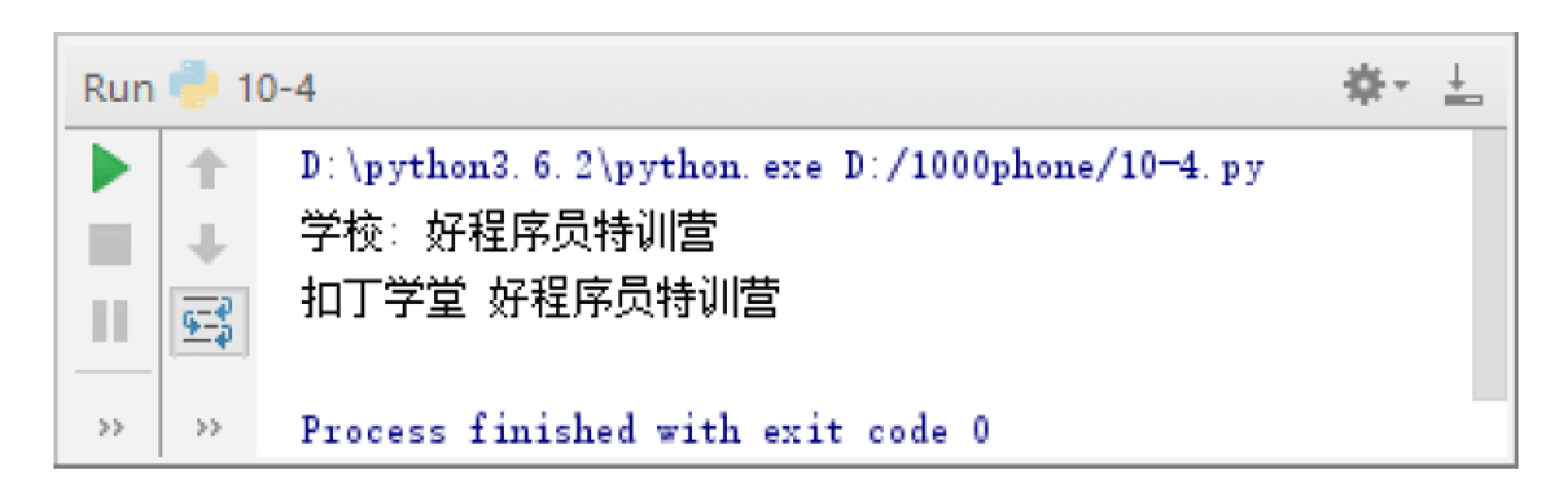


图 10.7 例 10-4 运行结果

例 10-5 通过赋值运算符修改或增加类对象与实例对象的属性。

```
1   class Student:
2       school = '扣丁学堂'          # 类属性
3       def say(self, name):        # 实例方法
4           self.name = name        # 实例属性
5           print('我是', self.name)
6   s1 = Student()                  # 创建实例对象
7   s1.say('小千')
8   s1.name = '小锋'
9   s1.age = 18
10  Student.school = '千锋教育'
11  print(Student.school, s1.name, s1.age)
```

运行结果如图 10.8 所示。

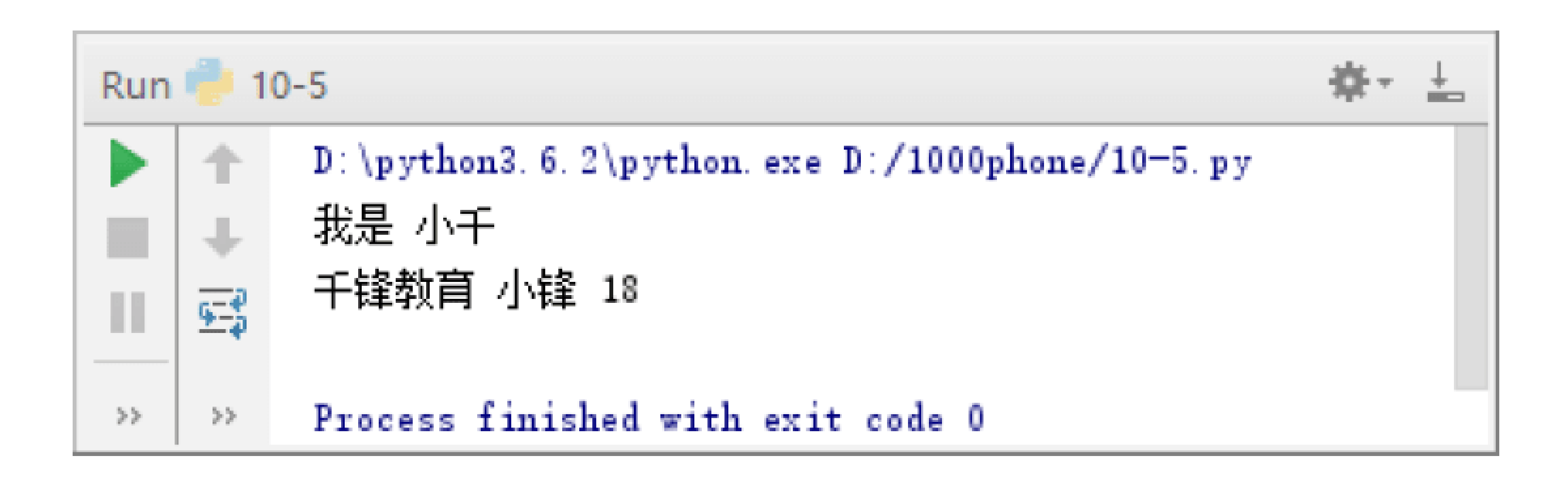


图 10.8 例 10-5 运行结果

在例 10-5 中，第 8 行修改实例属性 name 为'小锋'，第 9 行增加实例属性 age，第 10 行修改类属性 school 为'千锋教育'。

10.4 构造方法

Python 中构造方法一般用来为实例属性设置初值或进行其他必要的初始化操作，在创建实例对象时被自动调用和执行，如例 10-6 所示。

例 10-6 构造方法。

```
1  class Student:
2      def   init   (self):        # 构造方法
3          print('调用构造方法')
4          self.school = '千锋教育'
5      def say(self):
6          print('学校: ', self.school)
7  s1 = Student()                  # 创建实例对象
8  s1.say()
```

运行结果如图 10.9 所示。

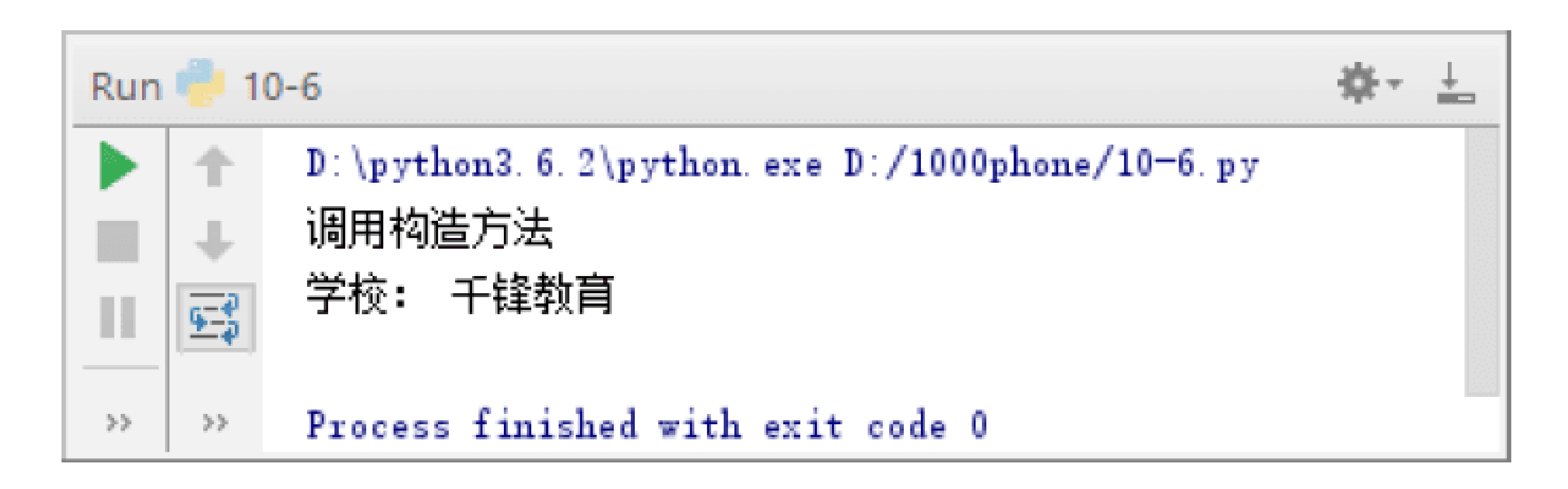


图 10.9 例 10-6 运行结果

在例 10-6 中，第 2 行定义一个构造方法，该方法名称（__init__）是固定不变的，如果用户没有定义它，Python 将提供一个默认的构造方法。第 7 行创建实例对象时将会自动调用构造方法给 school 属性赋值为'千锋教育'。第 6 行在实例方法 say()中访问 school 属性的值。

上例中创建实例对象时，默认给 school 属性赋值为'千锋教育'。如果在创建实例对象时，由用户指定 school 属性的值，则可以在构造方法中添加额外参数，如例 10-7 所示。

例 10-7 在构造方法中添加额外参数。

```
1  class Student:
2      def __init__(self, mySchool = '千锋教育'):  # 构造方法
3          self.school = mySchool
4      def say(self):
5          print('学校: ', self.school)
6  s1 = Student()
7  s1.say()
8  s2 = Student('扣丁学堂')
9  s2.say()
```

运行结果如图 10.10 所示。

在例 10-7 中，第 2 行定义一个构造方法，其中增加了一个默认参数。第 6 行创建实例对象 s1 调用构造方法时，使用默认值'千锋教育'。第 8 行创建实例对象 s2 调用构造方法时，使用指定的参数值'扣丁学堂'。

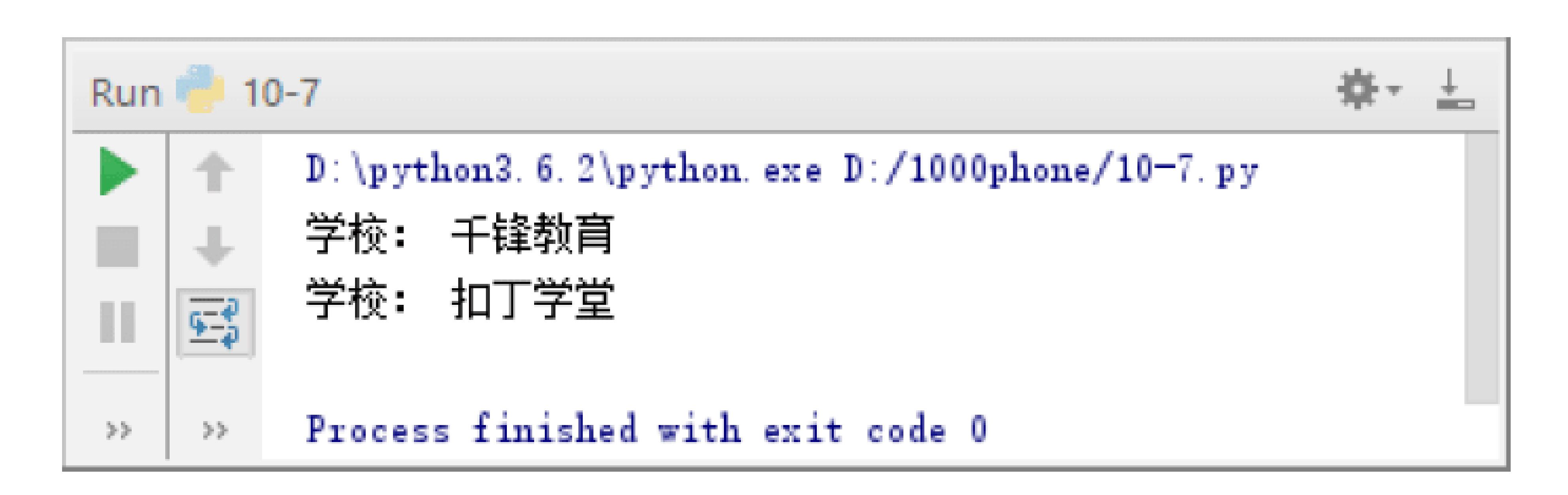


图 10.10　例 10-7 运行结果

10.5　析构方法

析构方法一般用来释放对象占用的资源，在删除对象和收回对象空间时被自动调用和执行，如例 10-8 所示。

例 10-8　析构方法。

```
1   class Student:
2       def __init__(self, myName):  # 构造方法
3           self.name = myName
4           print('初始化', self.name)
5       def __del__(self):           # 析构方法
6           print('释放对象占用资源', self.name)
7   s1 = Student('小千')
8   s2 = Student('小锋')
9   del s2
10  print('程序结束')
```

运行结果如图 10.11 所示。

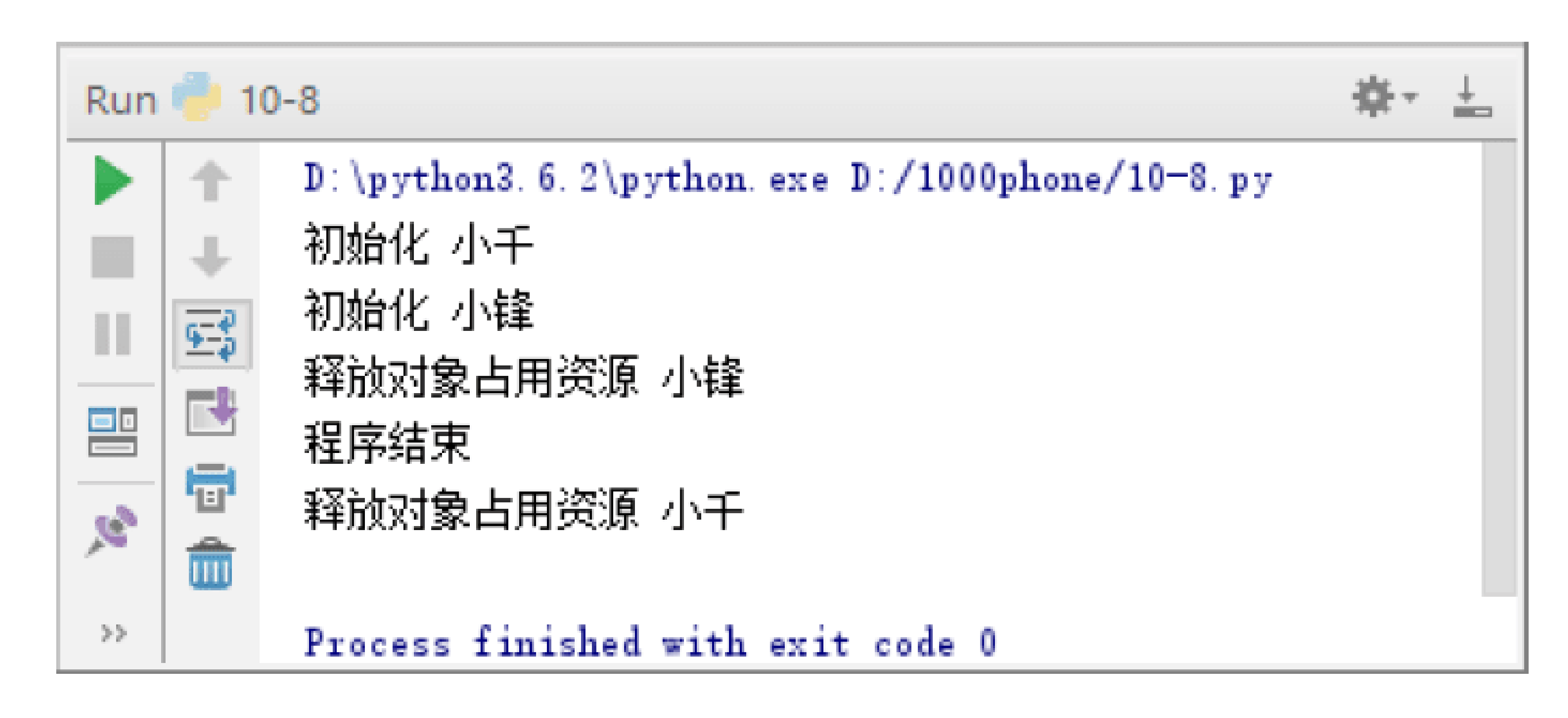


图 10.11　例 10-8 运行结果

在例 10-8 中，第 5 行定义一个析构方法，该方法名称（__del__）是固定不变的，如果用户没有定义它，Python 将提供一个默认的析构方法。第 9 行使用 del 语句删除一个对象，此时会自动调用析构方法。当程序结束时，Python 解释器会自动检测当前是否

存在未释放的对象，如果存在，则自动使用 del 语句释放其占用的内存，如本例中的 s1 对象。

10.6 类 方 法

类方法是类所拥有的方法，通过修饰器@classmethod 在类中定义，其语法格式如下：

```
class 类名:
    @classmethod
    def 类方法名(cls)
        方法体
```

其中，cls 表示类本身，通过它可以访问类的相关属性，但不可以访问实例属性，如例 10-9 所示。

例 10-9 类方法。

```
1   class Student:
2       num = 0
3       def __init__(self, myName):
4           Student.num += 1
5           self.name = myName
6       @classmethod   # 类方法
7       def count(cls):
8           print('学生个数:', cls.num)
9   Student.count()
10  s1 = Student('小千')
11  s1.count()
12  s2 = Student('小锋')
13  Student.count()
```

运行结果如图 10.12 所示。

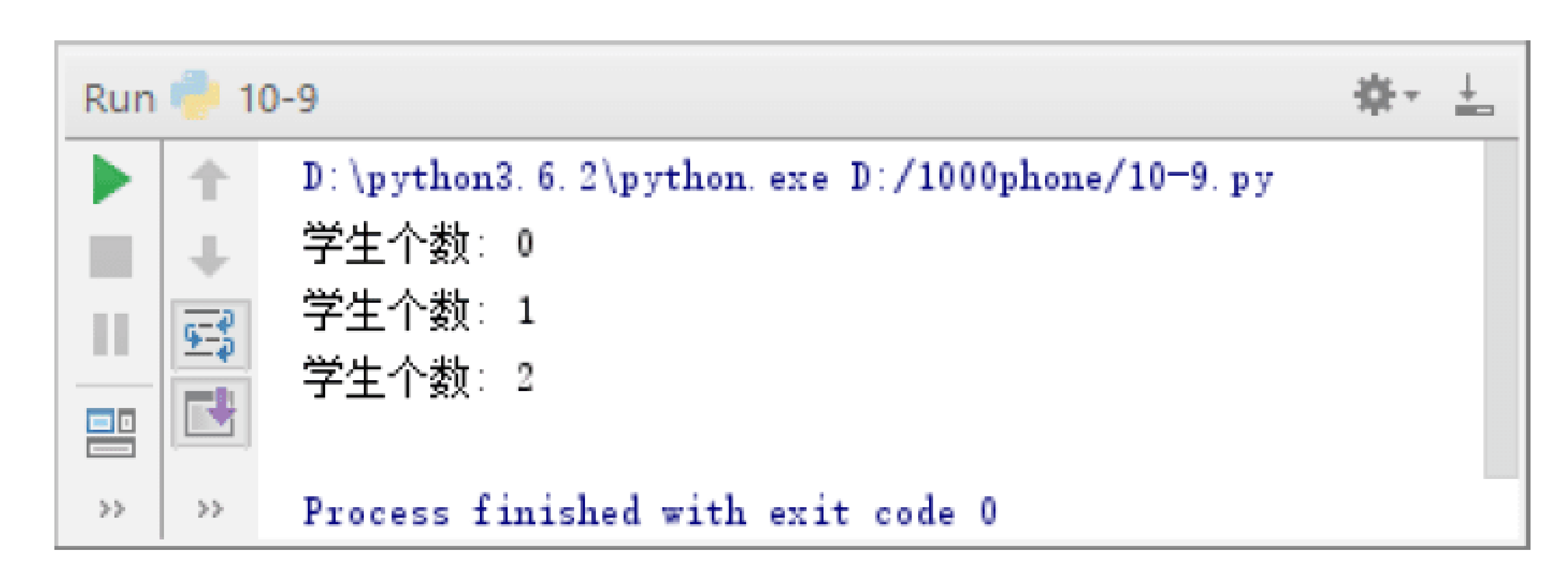


图 10.12 例 10-9 运行结果

在例 10-9 中，第 4 行在实例方法中通过“类名.类属性”的方式访问类属性 num。第 8 行在类方法中通过“cls.类属性”的方式访问类属性 num。第 9 行在创建实例对象之

前通过“类名.类方法名”调用类方法 count()。第 11 行通过“实例对象名.类方法名”调用类方法 count()。

10.7 静 态 方 法

类方法可以通过类名或实例对象名调用，静态方法也可以通过两者调用，其语法格式如下：

```
class 类名:
    @staticmethod
    def 静态方法名():
        函数体
```

其中，@staticmethod 为装饰器，参数列表中可以没有参数。静态方法可以访问类属性，但不可以访问实例属性，如例 10-10 所示。

例 10-10 静态方法。

```
1   class Student:
2       num = 0
3       def __init__(self, myName):
4           Student.num += 1
5           self.name = myName
6       @staticmethod    # 静态方法
7       def count():
8           print('学生个数:', Student.num)
9   Student.count()
10  s1 = Student('小千')
11  s1.count()
12  s2 = Student('小锋')
13  Student.count()
```

运行结果如图 10.13 所示。

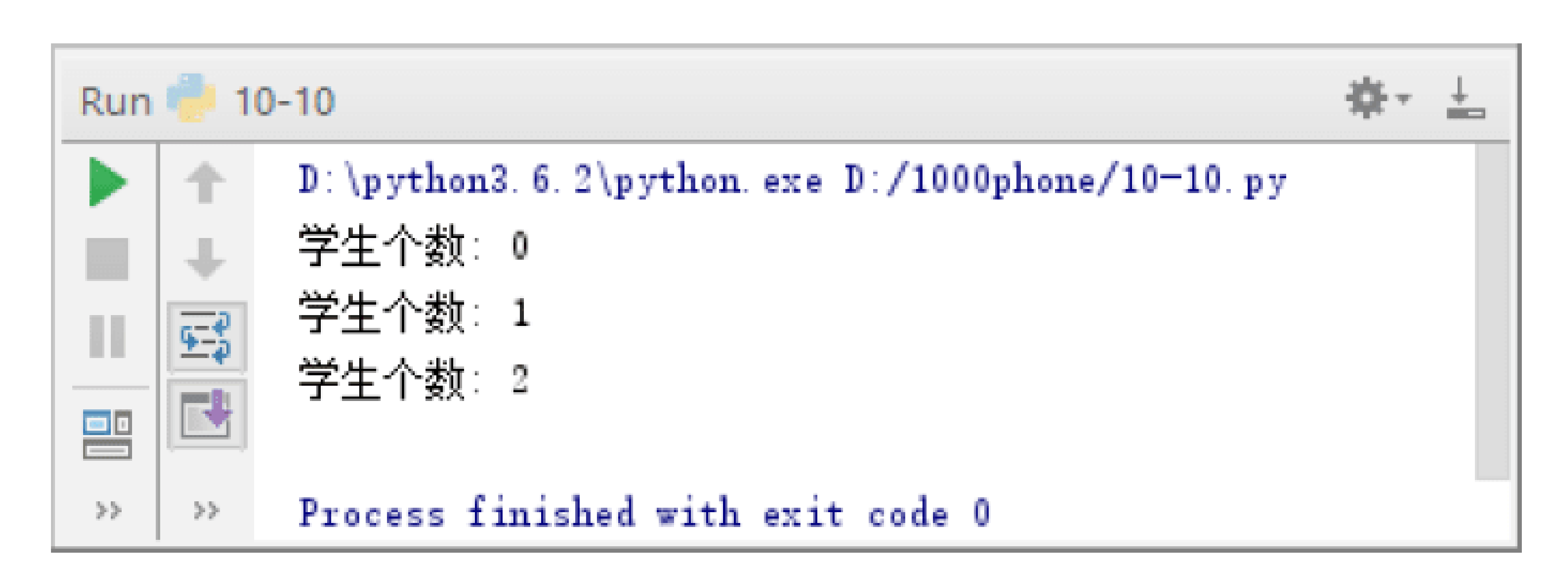


图 10.13 例 10-10 运行结果

在例 10-10 中，第 8 行在静态方法中通过“类名.类属性”的方式访问类属性 num。

第 9 行在创建实例对象之前通过“类名.静态方法名”调用静态方法 count()。第 11 行通过“实例对象名.静态方法名”调用静态方法 count()。

10.8 运算符重载

在 Python 中可通过运算符重载来实现对象之间的运算，如字符串可以进行如下运算：

```
'www.codingke' + '.com'
```

字符串可以通过“+”运算符实现字符串连接操作，其本质是通过__add__方法重载了运算符“+”，因此上述代码还可以写成如下代码：

```
'www.codingke'.__add__('.com')
```

Python 把运算符与类的实例方法关联起来，每个运算符都对应一个方法。运算符重载就是让类的实例对象可以参与内置类型的运算，表 10.1 列出了部分运算符重载方法。

表 10.1 部分运算符重载方法

运 算 符	方 法	说 明	示例（a、b 均为对象）
+	__add__(self, other)	加法	a + b
–	__sub__(self, other)	减法	a–b
*	__mul__(self, other)	乘法	a * b
/	__truediv__(self, other)	除法	a / b
%	__mod__(self, other)	求余	a % b
<	__lt__(self, other)	小于	a < b
<=	__le__(self, other)	小于或等于	a <= b
>	__gt__(self, other)	大于	a > b
>=	__ge__(self, other)	大于或等于	a >= b
==	__eq__(self, other)	等于	a == b
!=	__ne__(self, other)	不等于	a != b
[index]	__getitem__(self, item)	下标运算符	a[0]
in	__contains__(self, item)	检查是否是成员	r in a
len	__len__(self)	元素个数	len(a)
str	__str__(self)	字符串表示	str(a)

10.8.1 算术运算符重载

定义一个复数类并对其进行算术运算符重载，如例 10-11 所示。

例 10-11 算术运算符重载。

```
class MyComplex: # 定义复数类
    def __init__(self, r = 0, i = 0):    # 构造方法
```

```
        self.r = r # 实部
        self.i = i # 虚部
    def __add__(self, other):          # 重载加运算
        return MyComplex(self.r + other.r, self.i + other.i)
    def __sub__(self, other):          # 重载减运算
        return MyComplex(self.r - other.r, self.i - other.i)
    def __mul__(self, other):          # 重载乘运算
        return MyComplex(self.r*other.r - self.i*other.i,
                        self.i*other.r + self.r*other.i)
    def __truediv__(self, other):      # 重载除运算
        return MyComplex(
            (self.r*other.r + self.i*other.i)/(other.r**2 + other.i**2),
            (self.i*other.r - self.r*other.i)/(other.r**2 + other.i**2))
    def show(self):                    # 显示复数
        if self.i < 0:
            print('(', self.r, self.i, 'j)', sep = '')
        else:
            print('(', self.r, '+', self.i, 'j)', sep = '')
c1 = MyComplex(6, -8)
c2 = MyComplex(3, 4)
(c1 + c2).show()
(c1 - c2).show()
(c1 * c2).show()
(c1 / c2).show()
```

运行结果如图 10.14 所示。

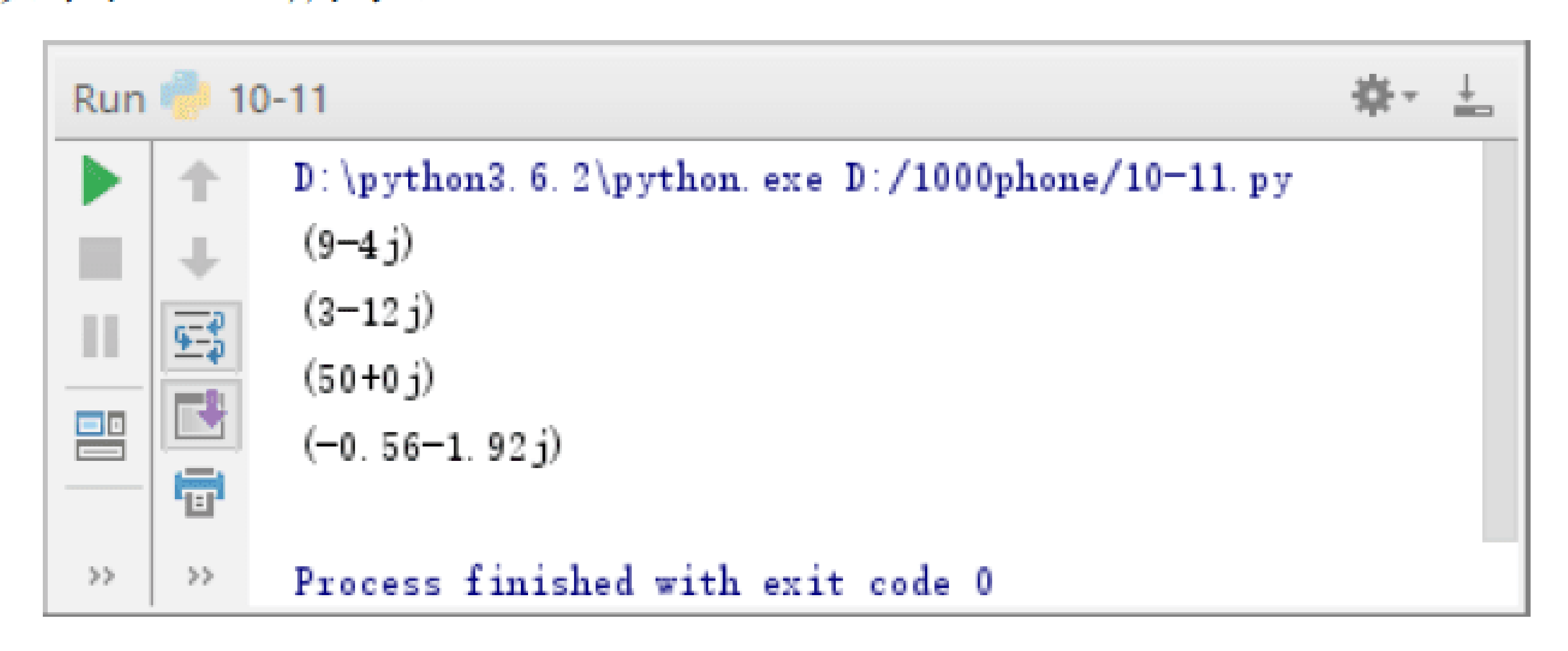


图 10.14　例 10-11 运行结果

在例 10-11 中，定义了一个 MyComplex 类，通过__add__()、__sub__()、__mul__()、__truediv__()方法分别重载+、−、*、/运算符。

10.8.2　比较运算符重载

定义一个复数类并对其进行比较运算符重载，如例 10-12 所示。

例 10-12 比较运算符重载。

```
class MyComplex: # 定义复数类
    def __init__(self, r = 0, i = 0):    # 构造方法
        self.r = r # 实部
        self.i = i # 虚部
    def __eq__(self, other):             # 重载==运算符
        return self.r == other.r and self.i == other.i
c1 = MyComplex(6, -8)
c2 = MyComplex(6, -8)
print(c1 == c2)
print(c1 != c2)
```

运行结果如图 10.15 所示。

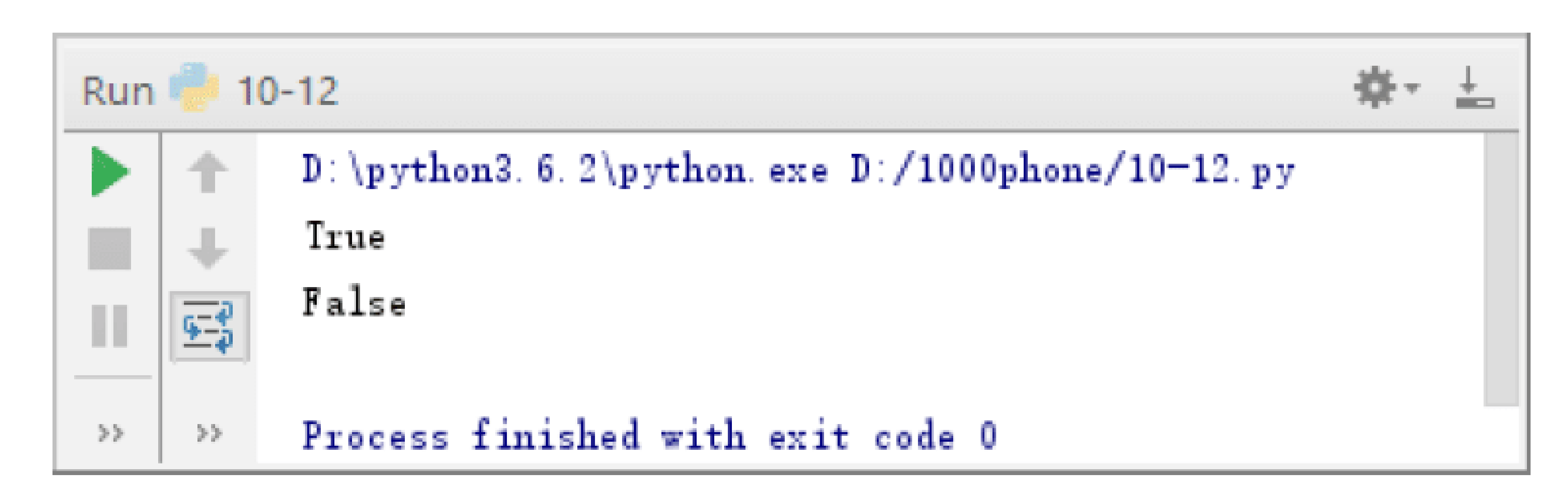


图 10.15 例 10-12 运行结果

在例 10-12 中，定义了一个 MyComplex 类，通过__eq__ ()方法重载==运算符。

10.8.3 字符串表示重载

当对象作为print()、str()函数的参数时，该对象会调用重载的__str__()方法，如例 10-13 所示。

例 10-13 字符串表示重载。

```
class MyComplex: # 定义复数类
    def __init__(self, r = 0, i = 0): # 构造方法
        self.r = r # 实部
        self.i = i # 虚部
    def __str__(self): # 重载__str__()
        if self.i < 0:
            return '(' + str(self.r) + str(self.i)+'j)'
        else:
            return '(' + str(self.r) + '+' + str(self.i) + 'j)'
c1 = MyComplex(6, -8)
c2 = MyComplex(6, 8)
print(c1, str(c2))
```

运行结果如图 10.16 所示。

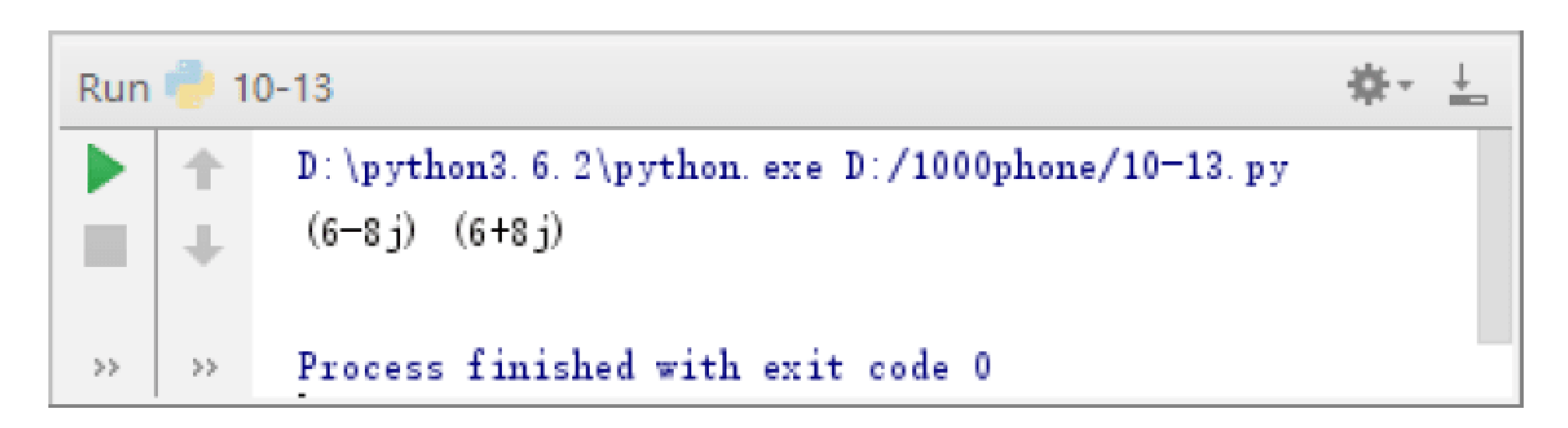


图 10.16 例 10-13 运行结果

在例 10-13 中，定义了一个 MyComplex 类，通过__str__()方法重载字符串表示。

10.8.4 索引或切片重载

当对实例对象执行索引、切片或 for 迭代时，该对象会调用重载的__getitem__()方法，如例 10-14 所示。

例 10-14 重载__getitem__()方法。

```
class Data:                              # 定义 Data 类
    def __init__(self, list):            # 构造方法
        self.data = list[:]
    def __getitem__(self, item):         # 重载索引与切片
        return self.data[item]
data = Data([1, 2, 3, 4])
print(data[2])
print(data[1:])
for i in data:
    print(i, end = ' ')
```

运行结果如图 10.17 所示。

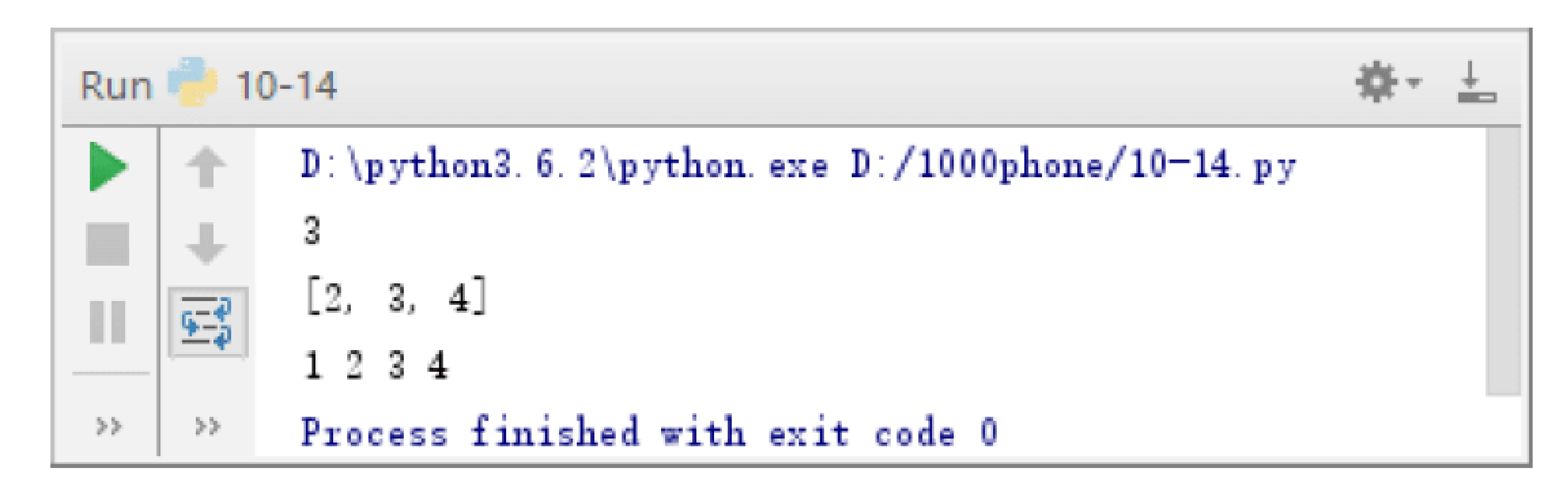


图 10.17 例 10-14 运行结果

在例 10-14 中，定义了一个 Data 类，通过__getitem__()方法重载索引与切片。

此外，在通过赋值语句给索引或切片赋值时，实例对象将调用__setitem__()方法实现对序列对象的修改，如例 10-15 所示。

例 10-15 重载__setitem__()方法。

```
class Data:                                  # 定义 Data 类
    def __init__(self, list):                # 构造方法
        self.data = list[:]
    def __setitem__(self, key, value):       # 重载索引与切片赋值
        self.data[key] = value
data = Data([1, 2, 3, 4])
print(data.data)
data[1] = '千锋教育'
data[2:] = '扣丁学堂', '好程序员特训营'
print(data.data)
```

运行结果如图 10.18 所示。

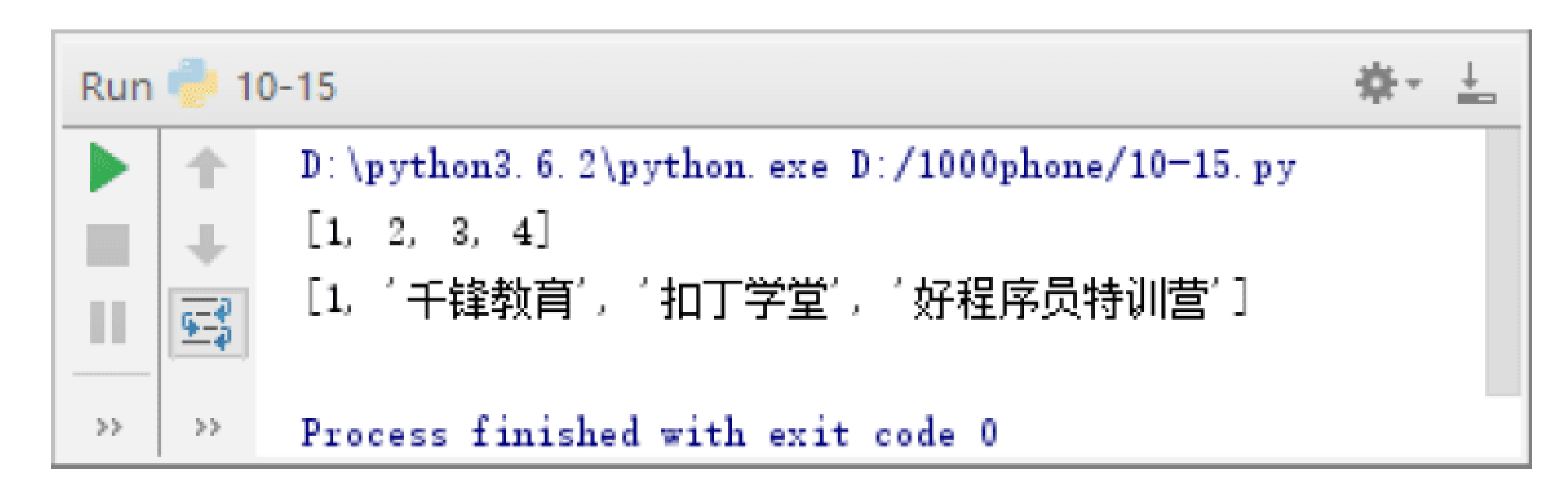


图 10.18 例 10-15 运行结果

在例 10-15 中，定义了一个 Data 类，通过__ setitem__()方法重载索引或切片赋值。

10.8.5 检查成员重载

当对实例对象执行检查成员时，该对象会调用重载的__contains__()方法，如例 10-16 所示。

例 10-16 检查成员重载。

```
class Data:                              # 定义 Data 类
    def __init__(self, list):            # 构造方法
        self.data = list[:]
    def __contains__(self, item):        # 重载__contains__()
        return item in self.data
data = Data([1, 2, 3, 4])
print(1 in data)
print(0 in data)
```

运行结果如图 10.19 所示。

在例 10-16 中，定义了一个 Data 类，通过__contains__()方法重载检测成员运算符。

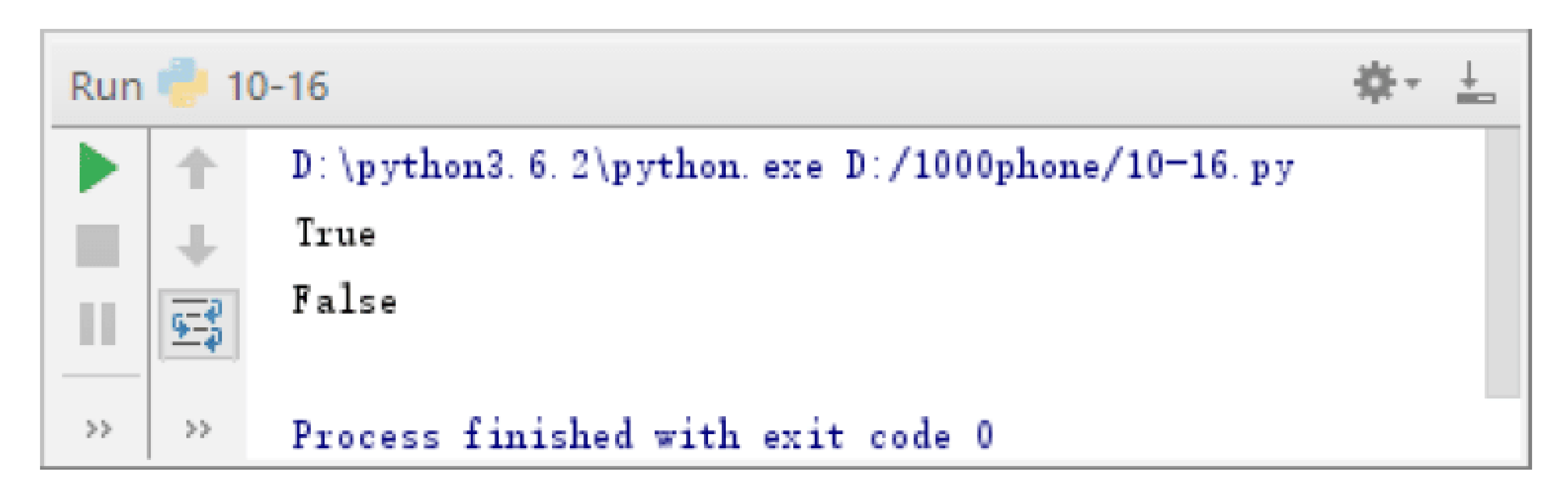


图 10.19　例 10-16 运行结果

10.9 小　案　例

通过扑克牌类与玩家类设计扑克牌发牌程序，要求程序随机将 52 张牌（不含大小王）发给 4 位玩家，最终在屏幕上显示每位玩家的牌，具体实现如例 10-17 所示。

例 10-17　随机将 52 张牌发给 4 位玩家。

```
# 定义牌类
class Card:
    # 牌面数字:1-13
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    # 牌面图标:红心、方块、梅花、黑桃
    SUITS = ["♥", "♦", "♣", "♠"]
    # 构造方法
    def __init__(self):
        self.cards = []
    # 生成牌
    def create(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.cards.append((suit, rank))
        self.shuffle()
    # 洗牌
    def shuffle(self):
        import random
        random.shuffle(self.cards)
    # 发牌,每位玩家默认 13 张牌
    def deal(self, players, perCards = 13):
        for rounds in range(perCards):
            for player in players:
                    topCard = self.cards[0]
                    self.cards.remove(topCard)
                    player.add(topCard)
```

```
# 定义玩家类
class Player:
    # 构造方法
    def __init__(self, name):
        self.name = name
        self.cards = []
    # 重载字符串表示方法
    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card[0])+ str(card[1]) + "  "
        else:
            rep = "无牌"
        return rep
    # 添加牌
    def add(self, card):
        self.cards.append(card)
# 测试
if __name__ == "__main__":
    # 4 位玩家
    players = [Player('小千'), Player('小锋'),
               Player('小扣'), Player('小丁')]
    # 生成一副牌
    card = Card()
    card.create()
    # 发给每位玩家 13 张牌
    card.deal(players, 13)
    # 显示 4 位玩家的牌
    for player in players:
        print(player.name, "玩家:", player)
```

运行结果如图 10.20 所示。

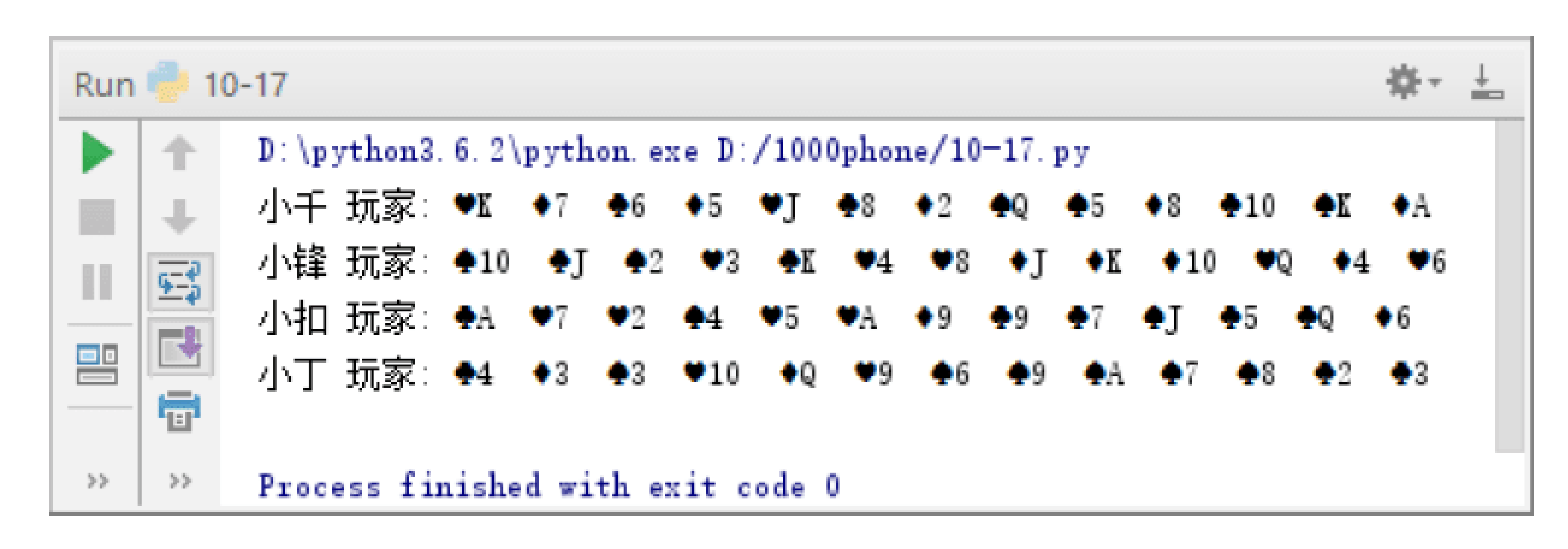


图 10.20 例 10-17 运行结果

在例 10-17 中，定义了一个 Card 类与 Player 类，两者通过 Card 类中的实例方法 deal()完成扑克牌的发放。

10.10 本章小结

本章主要介绍了 Python 中面向对象的基本概念，包括类的定义、对象的创建、构造方法与析构方法、类方法与静态方法、运算符重载。在编写大型应用程序时，应考虑使用面向对象的思想进行编程。

10.11 习　题

1．填空题

（1）使用________关键字定义类。
（2）类的所有实例方法都必须至少有一个名为________的参数。
（3）实例对象通过调用________来创建。
（4）使用 del 语句删除一个对象，程序会自动调用________方法。
（5）类方法通过修饰器________在类中定义。

2．选择题

（1）构造方法名称为（　　）。
A．__del__　　B．__doc__
C．__init__　　D．__str__
（2）当对象作为 print()函数的参数时，该对象会调用重载的（　　）方法。
A．__del__()　　B．__init__()
C．__getitem__()　　D．__str__()
（3）（　　）用于标识静态方法。
A．@classmethod　　B．@staticmethod
C．@instancemethod　　D．@objectmethod
（4）一般将（　　）作为类方法的第一个参数名称。
A．cls　　B．this
C．self　　D．类名
（5）实例属性只能通过（　　）访问。
A．实例对象名　　B．类对象名
C．类名　　D．上述 3 项

3．思考题



（1）简述类对象与实例对象的区别。
（2）简述实例方法、类方法、静态方法的区别。

4．编程题

编写程序，定义一个 Point（点）类，使其能完成打印功能和统计创建点对象个数功能。

第 11 章 chapter 11

面向对象（下）

本章学习目标

- 理解面向对象的三大特征。
- 掌握继承。
- 掌握多态。
- 了解设计模式。

第 10 章讲解了面向对象中类与对象的基本概念，本章主要讲解面向对象的三大特征：封装、继承、多态，面向对象的灵活应用可以增强代码的安全性、重用性及可维护性。

11.1 面向对象的三大特征

面向对象程序设计实际上就是对现实世界的对象进行建模操作。面向对象程序设计的特征主要可以概括为封装、继承和多态，接下来针对这 3 种特性进行简单介绍。

1. 封装

封装是面向对象程序设计的核心思想。它是指将对象的属性和行为封装起来，其载体就是类，类通常对客户隐藏其实现细节，这就是封装的思想。例如，计算机的主机是由内存条、硬盘、风扇等部件组成，生产厂家把这些部件用一个外壳封装起来组成主机，用户在使用该主机时，无须关心其内部的组成及工作原理，如图 11.1 所示。

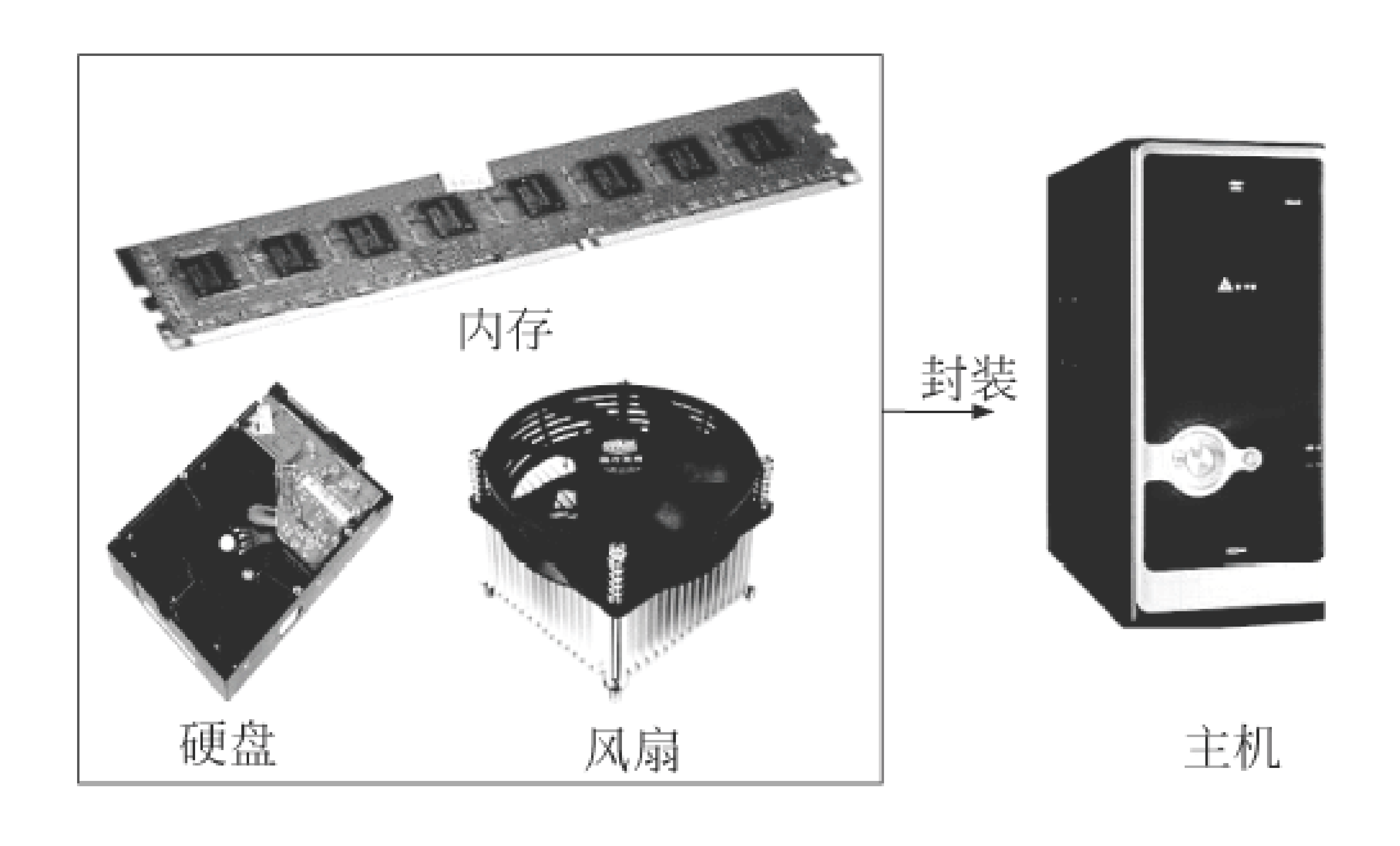


图 11.1 主机及其组成部件

2．继承

继承是面向对象程序设计提高重用性的重要措施。它体现了特殊类与一般类之间的关系，当特殊类包含了一般类的所有属性和行为，并且特殊类还可以有自己的属性和行为时，称作特殊类继承了一般类。一般类又称为父类或基类，特殊类又称为子类或派生类。例如，已经描述了汽车模型这个类的属性和行为，如果需要描述一个小轿车类，只需让小轿车类继承汽车模型类，然后再描述小轿车类特有的属性和行为，而不必再重复描述一些在汽车模型类中已有的属性和行为，如图 11.2 所示。

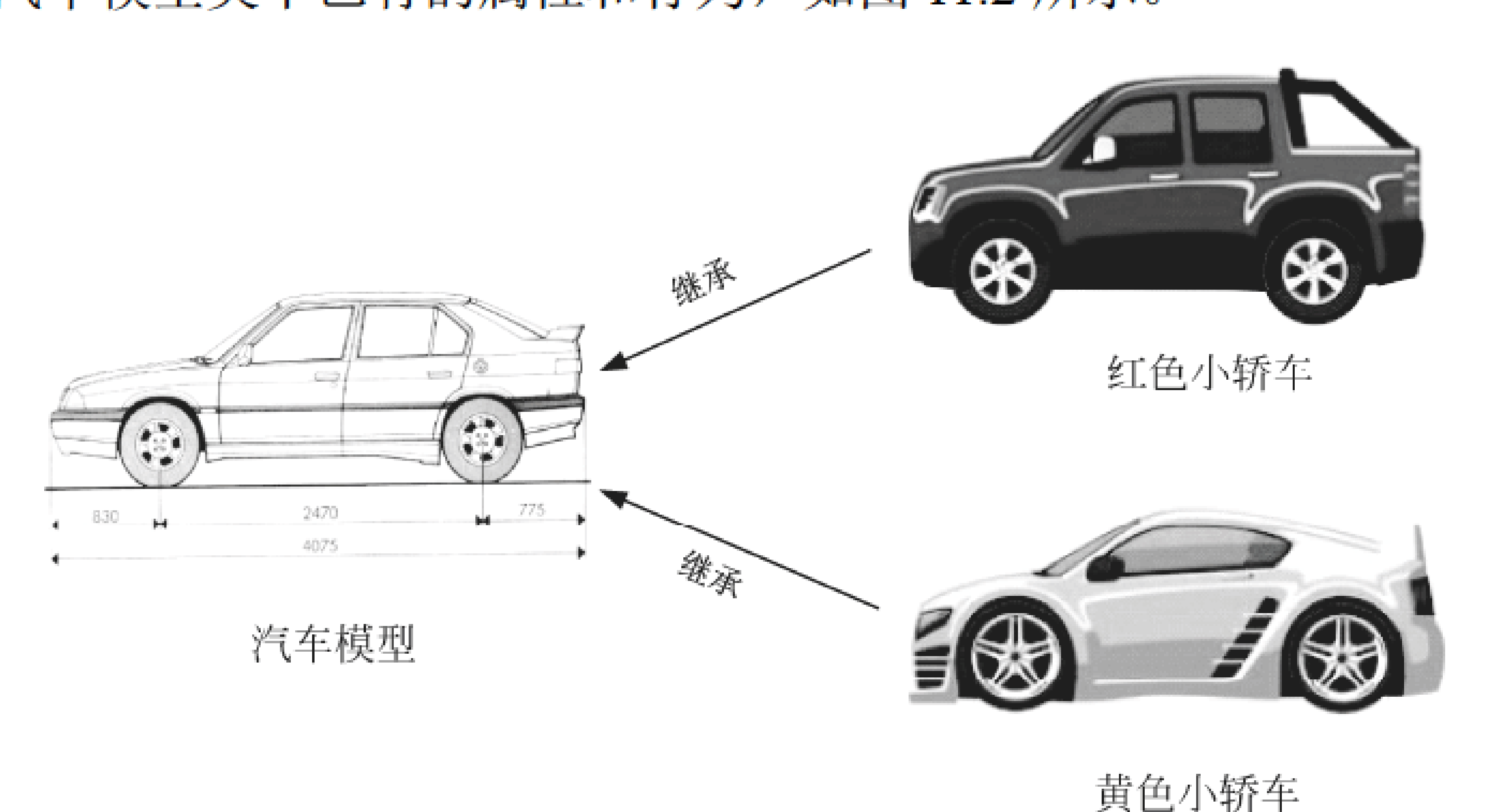


图 11.2　汽车模型与小轿车

3．多态

多态是面向对象程序设计的重要特征。生活中也常存在多态，例如，学校的下课铃声响了，这时有学生去买零食、有学生去打球、有学生在聊天。不同的人对同一事件产生了不同的行为，这就是多态在日常生活中的表现。程序中的多态是指一种行为对应着多种不同的实现。例如，在一般类中说明了一种求几何图形面积的行为，这种行为不具有具体含义，因为它并没有确定具体几何图形，又定义一些特殊类，如三角形、正方形、梯形等，它们都继承自一般类。在不同的特殊类中都继承了一般类的求面积的行为，可以根据具体的不同几何图形使用求面积公式，重新定义求面积行为的不同实现，使之分别实现求三角形、正方形、梯形等面积的功能，如图 11.3 所示。

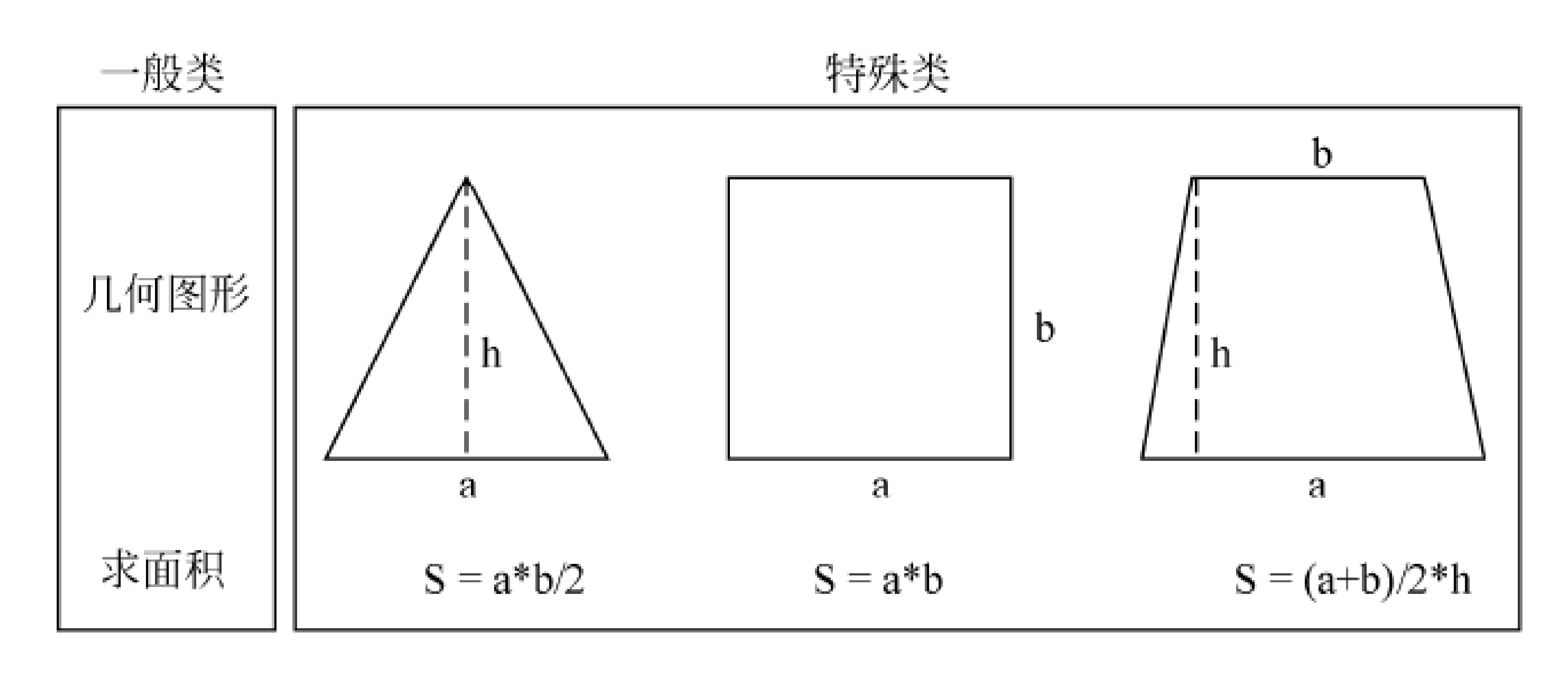


图 11.3　一般类与特殊类

在实际编写应用程序时，开发者需要根据具体应用设计对应的类与对象，然后在此基础上综合考虑封装、继承与多态，这样编写出的程序更健壮、更易扩展。

11.2 封　　装

类的封装可以隐藏类的实现细节，迫使用户只能通过方法去访问数据，这样就可以增强程序的安全性。接下来演示未使用封装可能出现的问题，如例 11-1 所示。

例 11-1　未使用封装。

```
class Student:
    def __init__(self, myName, myScore):
        self.name, self.score = myName, myScore
    def __str__(self):
        return '姓名:' + str(self.name) + '\t 成绩:' + str(self.score)
s1 = Student('小千', 100)
print(s1)
s1.score = -68
print(s1)
```

运行结果如图 11.4 所示。

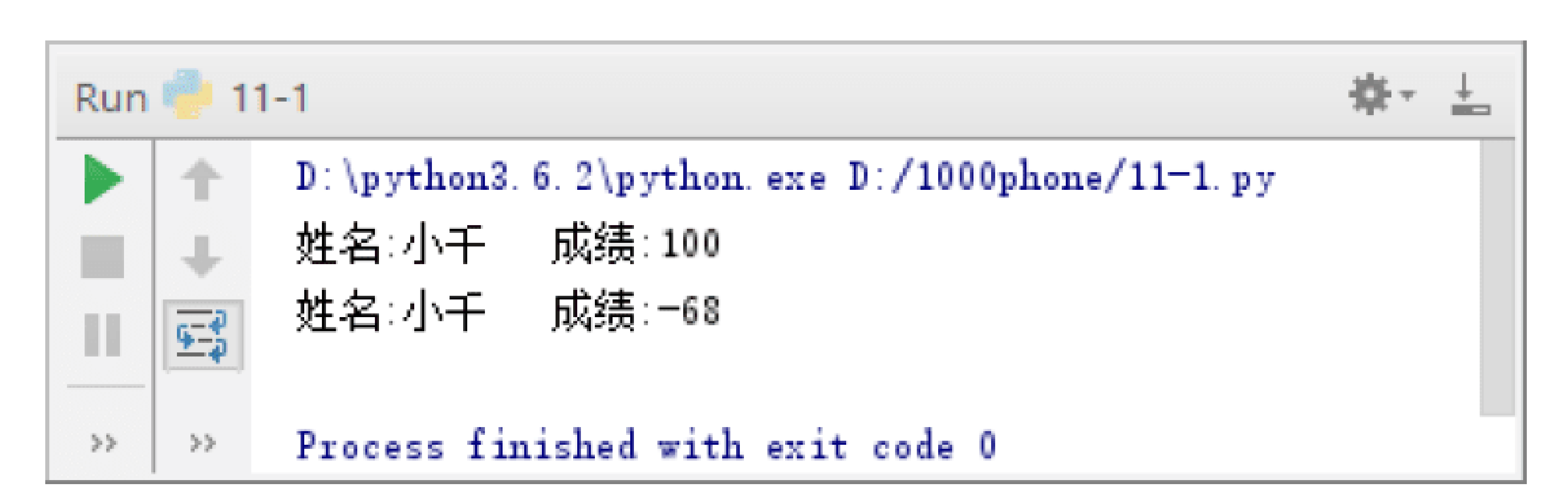


图 11.4　例 11-1 运行结果

在例 11-1 中，运行结果输出的成绩为−68，在程序中不会有任何问题，但在现实生活中明显是不合理的。为了避免这种不合理的情况，就需要用到封装，即不让使用者随意修改类的内部属性。在定义类时，可以将属性定义为私有属性，这样外界就不能随意修改。Python 中通过在属性名前加两个下画线来表明私有属性，如例 11-2 所示。

例 11-2　私有属性。

```
class Student:
    def __init__(self, myName, myScore):
        self.name, self.__score = myName, myScore
    def __str__(self):
        return '姓名:' + str(self.name) + '\t 成绩:' + str(self.__score)
s1 = Student('小千', 100)
```

```
7    print(s1)
8    s1.__score = -68
9    print(s1)
```

运行结果如图 11.5 所示。

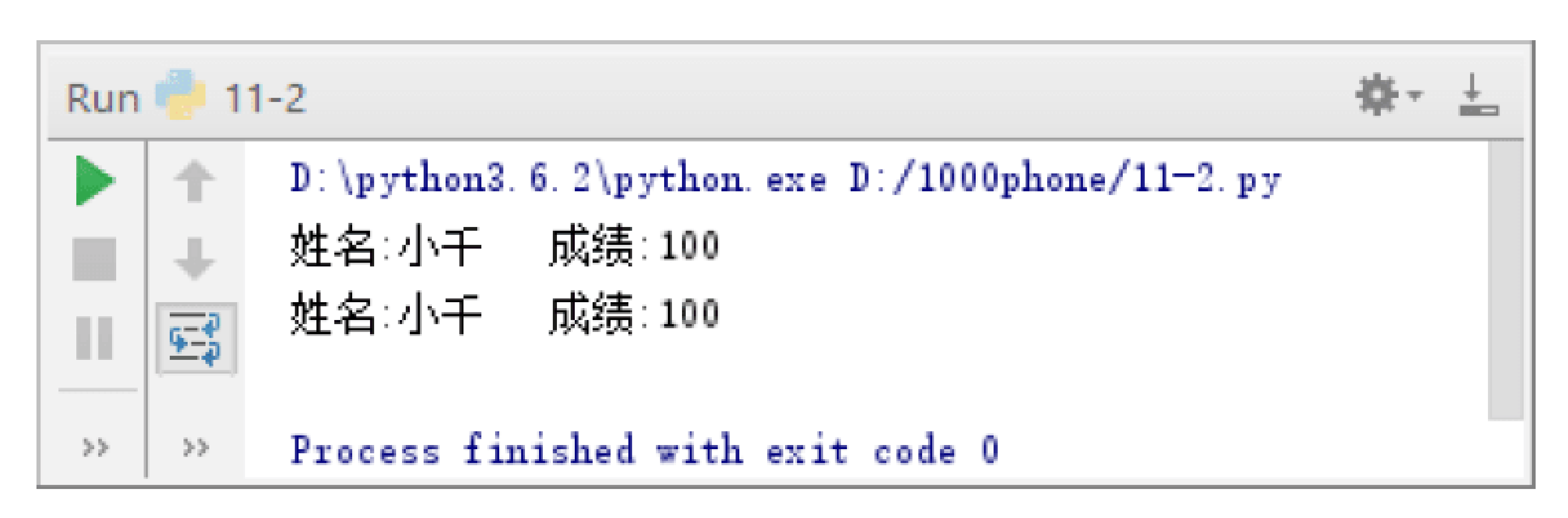


图 11.5　例 11-2 运行结果

在例 11-2 中，self.name 为公有属性，self.__score 为私有属性。第 8 行试图修改私有属性的值。从程序运行结果可看出，私有属性的值并没有发生变化。

当属性设置为私有属性后，经常需要提供设置或获取属性值的两个方法供外界使用，如例 11-3 所示。

例 11-3　设置或获取属性值的方法。

```
1    class Student:
2        def   init  (self, myName, myScore = 0):
3            self.name = myName
4            self.setScore(myScore)
5        def setScore(self, myScore):
6            if 0 < myScore <= 100:
7                self.__score = myScore
8            else:
9                self.  score = 0
10               print(self.name,'成绩有误!')
11       def getScore(self):
12           return self.  score
13       def __str__(self):
14           return '姓名:' + str(self.name) + '\t 成绩:' + str(self.__score)
15   s1 = Student('小千', -68)
16   print(s1)
17   s1.setScore(100)
18   print(s1.getScore(), s1)
```

运行结果如图 11.6 所示。

在例 11-3 中，第 17 行通过 setScore()方法设置私有属性 self.__score 的值，第 18 行通过 getScore()方法获取私有属性 self.__score 的值。

此外，私有属性在类外不能直接访问，但程序在测试或调试环境中，可以通过“对

象名._类名”的方式在类外访问，如例 11-4 所示。

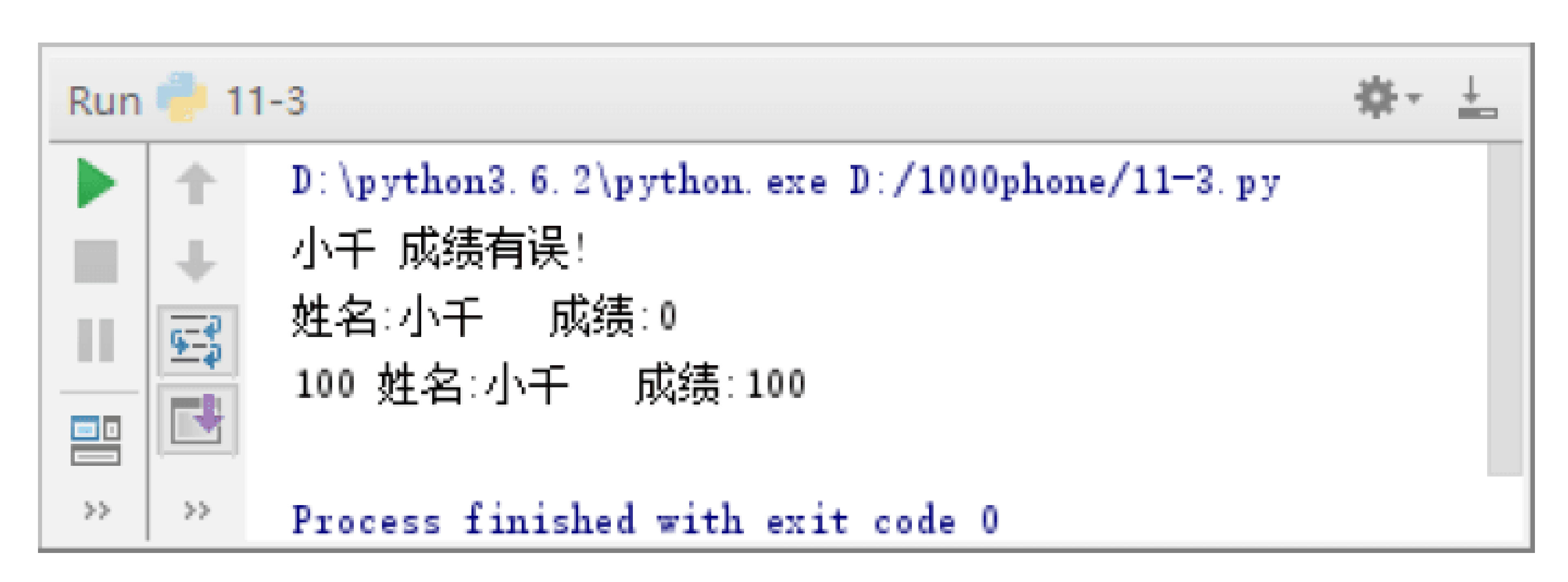


图 11.6　例 11-3 运行结果

例 11-4　在类外访问私有属性。

```
1   class Student:
2       def __init__(self, myName, myScore = 0):
3           self.name = myName
4           self.setScore(myScore)
5       def setScore(self, myScore):
6           if 0 < myScore <= 100:
7               self.  score = myScore
8           else:
9               self.  score = 0
10              print(self.name,'成绩有误!')
11      def getScore(self):
12          return self.  score
13      def   str   (self):
14          return '姓名:' + str(self.name) + '\t 成绩:' + str(self.__score)
15  s1 = Student('小千', 100)
16  print(s1)
17  s1._Student__score = 90
18  print(s1)
```

运行结果如图 11.7 所示。

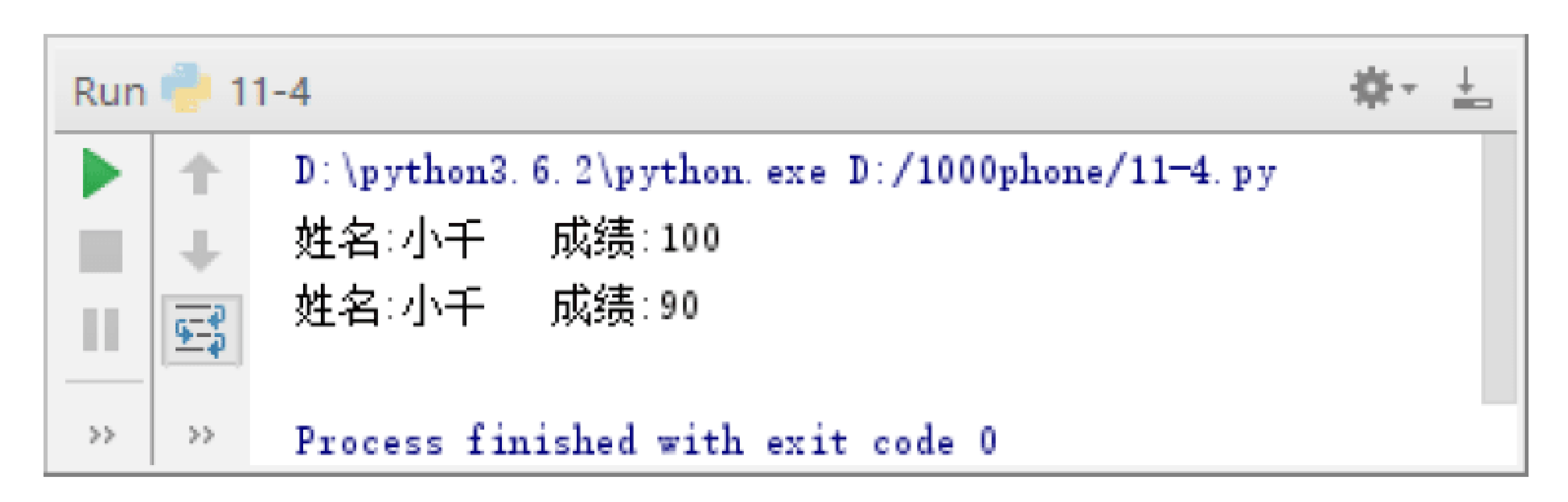


图 11.7　例 11-4 运行结果

在例 11-4 中，第 17 行通过“对象名._类名”的方式在类外修改私有属性的值。

11.3 继　　承

在自然界中，继承这个概念非常普遍，例如，熊猫宝宝继承了熊猫爸爸和熊猫妈妈的特性，它有着圆圆的脸颊、大大的黑眼圈、胖嘟嘟的身体，人们不会把它错认为是狒狒。在程序设计中，继承是面向对象的另一大特征，它用于描述类的所属关系，多个类通过继承形成一个关系体系。

11.3.1 单一继承

单一继承是指生成的派生类只有一个基类，如学生与教师都继承自人，如图 11.8 所示。

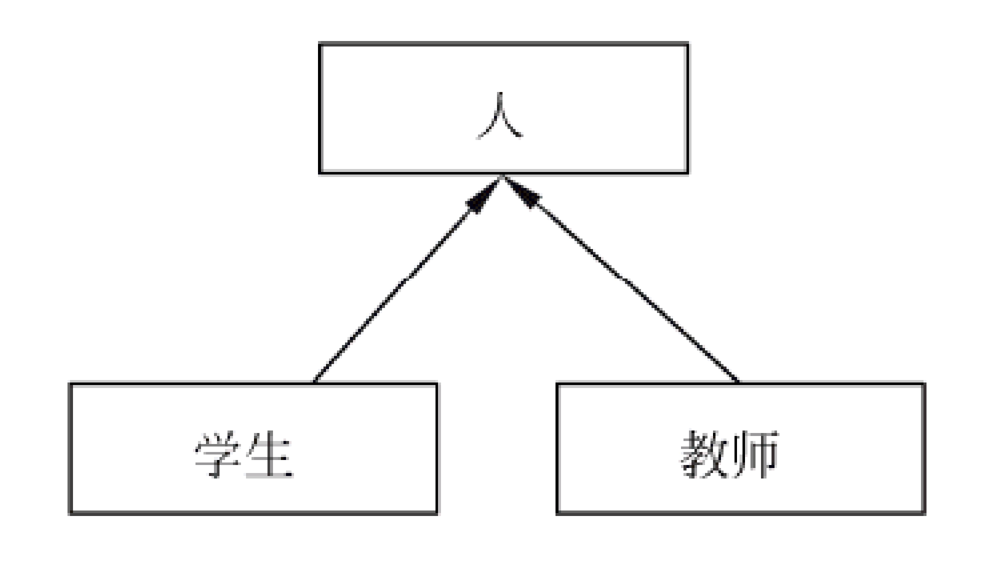


图 11.8　单一继承

单一继承由于只有一个基类，继承关系比较简单，操作比较容易，因此使用相对较多，其语法格式如下：

```
class 基类名(object):   # 等价于 class 基类名:
    类体
class 派生类名(基类名):
    类体
```

上述代码表示派生类继承自基类，派生类可以使用基类的所有公有成员，也可以定义新的属性和方法，从而完成对基类的扩展。注意 Python 中所有的类都继承自 object 类，第 10 章中出现的类省略了 object。

接下来演示如何定义单一继承，如例 11-5 所示。

例 11-5　单一继承。

```
class Person(object):
    def __init__(self, name):
        self.name = name
    def show(self):
        print('姓名:', self.name)
class Student(Person):
    pass
```

```
8    s1 = Student('小千')
9    s1.show()
```

运行结果如图 11.9 所示。

```
Run 11-5
D:\python3.6.2\python.exe D:/1000phone/11-5.py
姓名: 小千

Process finished with exit code 0
```

图 11.9　例 11-5 运行结果

在例 11-5 中，Student 类继承自 Person 类，第 9 行使用派生类实例对象调用基类中的公有方法。

大家可能会有疑问，派生类的构造方法名与基类的构造方法名相同，创建派生类实例对象如何调用构造方法，接下来演示这种情形，如例 11-6 所示。

例 11-6　派生类的构造方法名与基类的构造方法名相同。

```
1    class Person(object):
2        def __init__(self, name):
3            print('Person 类构造方法')
4            self.name = name
5        def show(self):
6            print('姓名:', self.name)
7    class Student(Person):
8        pass
9    class Teacher(Person):
10       def __init__(self, name):
11           print('Teacher 类构造方法')
12   s1 = Student('小千')
13   t1 = Teacher('小锋')
```

运行结果如图 11.10 所示。

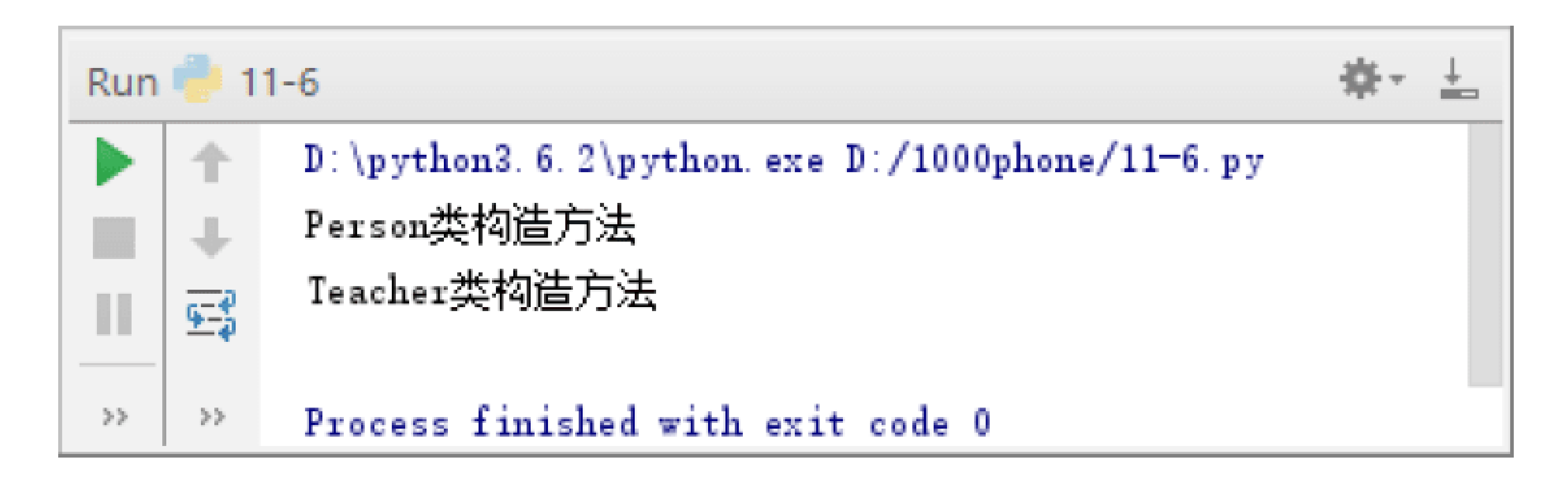


图 11.10　例 11-6 运行结果

在例 11-6 中，派生类 Student 中没有定义构造方法，第 12 行创建 Student 类实例对

象调用基类的构造方法；派生类 Teacher 中显式定义构造方法，第 13 行创建 Teacher 类实例对象调用自身的构造方法。

如果派生类的构造函数中需要添加参数，则可以在派生类的构造方法中调用基类的构造方法，如例 11-7 所示。

例 11-7 派生类的构造方法中调用基类的构造方法。

```
1   class Person(object):
2       def __init__(self, name):
3           print('Person 类构造方法')
4           self.name = name
5       def show(self):
6           print('姓名:', self.name)
7   class Student(Person):
8       def __init__(self, name, score):
9           print('Teacher 类构造方法')
10          super(Student, self).__init__(name)
11          self.__score = score
12      def __str__(self):
13          return '姓名:'+ str(self.name) + ' 分数:' + str(self.__score)
14  s1 = Student('小千', 100)
15  print(s1)
```

运行结果如图 11.11 所示。

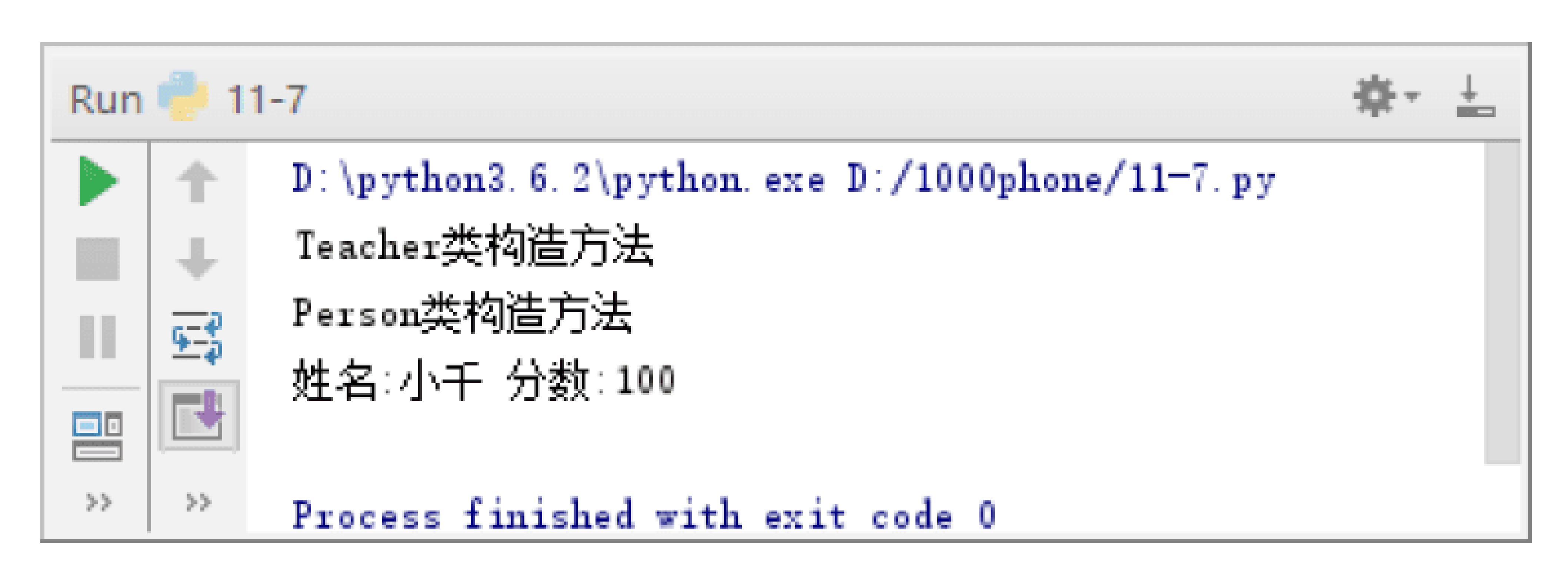


图 11.11 例 11-7 运行结果

在例 11-7 中，第 10 行通过 super()方法调用基类的构造方法，该行也可以写成如下两行中的任意一种形式，具体如下所示：

```
super().__init__(name)
Person.__init__(self, name)
```

如果派生类定义的属性和方法与基类的属性和方法同名，则派生类实例对象调用派生类中定义的属性和方法，如例 11-8 所示。

例 11-8 派生类实例对象调用派生类中定义的属性和方法。

```
class Person(object):
```

```
2       def   init   (self, name):
3           self.name = name
4       def show(self):
5           print('姓名:', self.name)
6   class Student(Person):
7       def   init   (self, name, score):
8           self.name, self.  score = name, score
9       def show(self):
10          print('姓名:', self.name, ' 分数:', self.  score)
11  s1 = Student('小千', 100)
12  s1.show()
```

运行结果如图 11.12 所示。

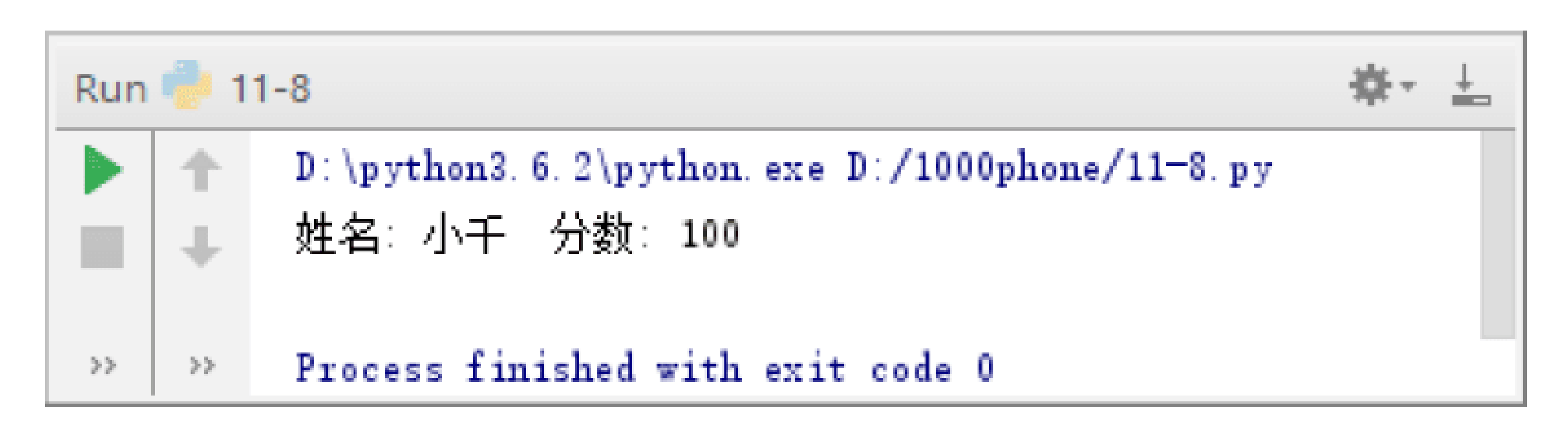


图 11.12　例 11-8 运行结果

在例 11-8 中，第 8 行在派生类中定义与基类实例属性名相同的属性 self.name，第 9 行在派生类中定义与基类实例方法名相同的方法 show()。从运行结果可看出，派生类实例对象 s1 调用派生类中定义的属性与方法。

另外，需特别注意，基类的私有属性和方法是不会被派生类继承的。因此，派生类不能访问基类的私有成员，如例 11-9 所示。

例 11-9　派生类不能访问基类的私有成员。

```
1   class Person(object):
2       def   init   (self, name):
3           self.__name = name
4       def   show(self):
5           print('姓名:', self.__name)
6   class Student(Person):
7       def test(self):
8           print(self.  name)
9           self.__show()
10  s1 = Student('小千')
11  s1.test()
```

运行结果如图 11.13 所示。

在例 11-9 中，第 8 行在派生类中访问基类中的私有属性__name。程序运行后报错，提示实例对象没有_Student__name 属性。当一个类中定义了私有成员，Python 会在该成

员名前添加“_类名”，因此错误中会提示没有_Student__name 属性。

```
Run 11-9
D:\python3.6.2\python.exe D:/1000phone/11-9.py
Traceback (most recent call last):
  File "D:/1000phone/11-9.py", line 11, in <module>
    s1.test()
  File "D:/1000phone/11-9.py", line 8, in test
    print(self.__name)
AttributeError: 'Student' object has no attribute
 '_Student__name'

Process finished with exit code 1
```

图 11.13　例 11-9 运行结果（一）

同理，将例 11-9 中的第 8 行代码注释掉，再次运行程序，运行结果如图 11.14 所示。

```
Run 11-9
D:\python3.6.2\python.exe D:/1000phone/11-9.py
Traceback (most recent call last):
  File "D:/1000phone/11-9.py", line 11, in <module>
    s1.test()
  File "D:/1000phone/11-9.py", line 9, in test
    self.__show()
AttributeError: 'Student' object has no attribute
 '_Student__show'

Process finished with exit code 1
```

图 11.14　例 11-9 运行结果（二）

11.3.2　多重继承

在现实生活中，在职研究生既是一名学生，又是一名职员。在职研究生同时具有学生和职员的特征，这种关系应用在面向对象程序设计上就是用多重继承来实现的，如图 11.15 所示。

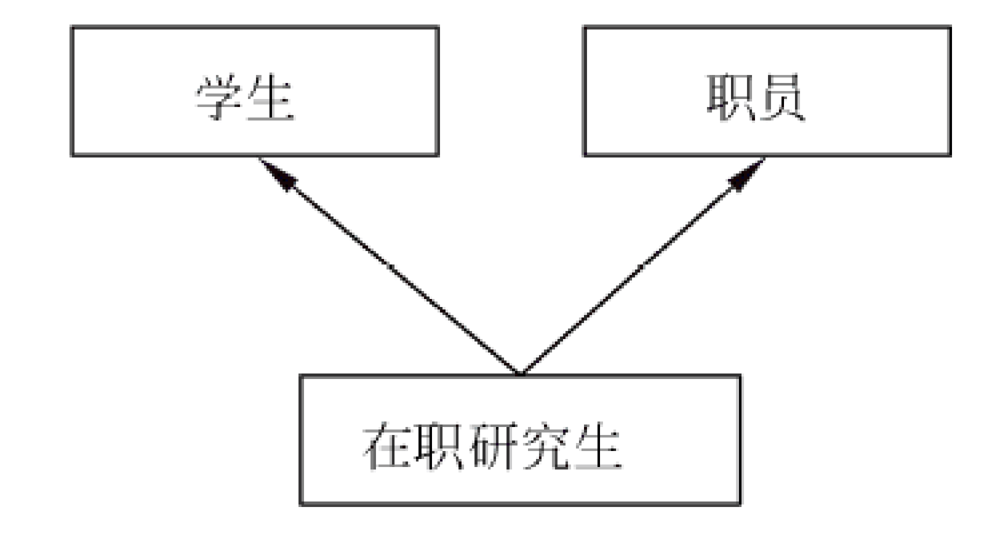


图 11.15　多重继承

多重继承指派生类可以同时继承多个基类，其语法格式如下：

```
class 基类1(object):
    类体
class 基类2(object):
    类体
class 派生类(基类1, 基类2):
    类体
```

上述代码表示派生类继承自基类 1 与基类 2。接下来演示如何定义多重继承，如例 11-10 所示。

例 11-10　多重继承。

```
1   class Student(object):
2       def __init__(self, name, score):
3           self.name, self.score = name, score
4       def showStd(self):
5           print('姓名:', self.name, ' 分数:', self.score)
6   class Staff(object):
7       def __init__(self, id, salary):
8           self.id, self.salary = id, salary
9       def showStf(self):
10          print('ID:', self.id, ' 薪资:', self.salary)
11  class OnTheJobGraduate(Student, Staff):
12      def   init   (self, name, score, id, salary):
13          Student.__init__(self, name, score)
14          Staff.__init__(self, id, salary)
15  g1 = OnTheJobGraduate('小千', 100, '110', 10000)
16  g1.showStd()
17  g1.showStf()
```

运行结果如图 11.16 所示。

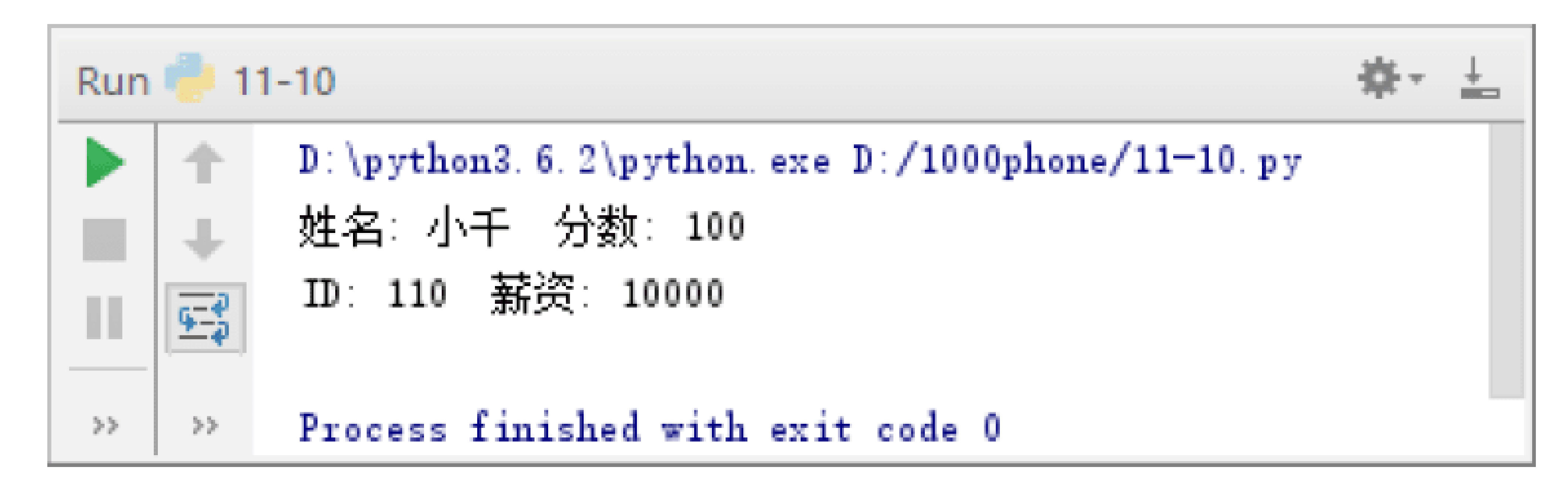


图 11.16　例 11-10 运行结果

在例 11-10 中，派生类 OnTheJobGraduate 继承自基类 Student 与 Staff，则派生类同时拥有两个基类的公有成员。第 12 行在派生类中定义构造方法并调用两个基类的构造方法。第 15 行创建派生类实例对象并对其属性进行初始化。

在多重继承中，如果基类存在同名的方法，Python 按照继承顺序从左到右在基类中搜索方法，如例 11-11 所示。

例 11-11 基类存在同名方法。

```
1   class Student(object):
2       def __init__(self, name, score):
3           self.name, self.score = name, score
4           print('Student类', self.name)
5       def show(self):
6           print('姓名:', self.name, ' 分数:', self.score)
7   class Staff(object):
8       def __init__(self, name, salary):
9           self.name, self.salary = name, salary
10          print('Staff类', self.name)
11      def show(self):
12          print('姓名:', self.name, ' 薪资:', self.salary)
13  class OnTheJobGraduate(Student, Staff):
14      def __init__(self, name1, score, name2, salary):
15          Student.__init__(self, name1, score)
16          Staff.__init__(self, name2, salary)
17  g1 = OnTheJobGraduate('小千', 100, 'xiaoqian', 10000)
18  g1.show()
```

运行结果如图 11.17 所示。

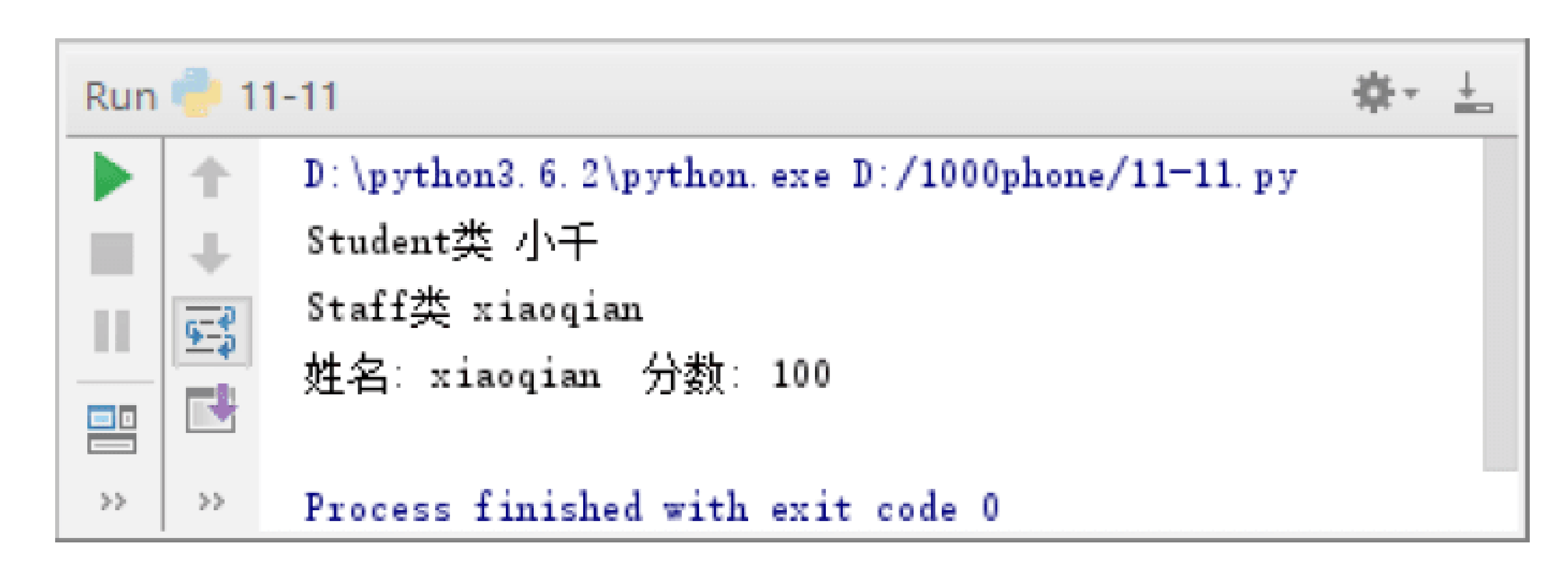


图 11.17 例 11-11 运行结果（一）

在例 11-11 中，基类 Student 与 Staff 中存在相同的属性名与方法名，第 18 行派生类实例对象调用基类 Student 中的 show()方法。此处需注意，该方法输出 self.name 为'xiaoqian'，而不是'小千'，因为派生类调用构造方法时，先调用 Student 类中的构造方法，此时 self.name 为'小千'，接着调用 Staff 类中的构造方法，此时会覆盖掉之前的内容，最终 self.name 为'xiaoqian'。

如果将例 11-11 中第 13 行代码中的 Student 与 Staff 交换位置，具体如下所示：

```
class OnTheJobGraduate(Staff, Student):
```

再次运行程序，则运行结果如图 11.18 所示。

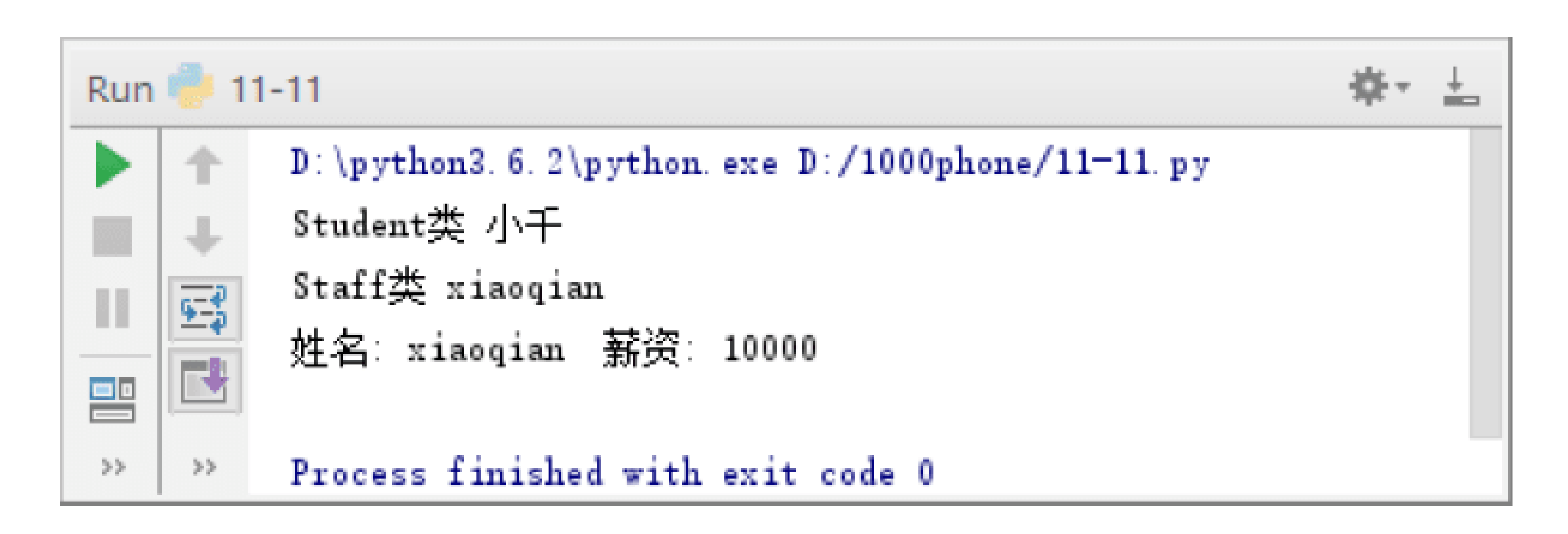


图 11.18　例 11-11 运行结果（二）

11.4　多　　态

Python 中加法运算符可以作用于两个整数，也可以作用于字符串，具体如下所示：

```
1 + 2              # 将整数 1 与 2 相加,结果为 3
'1' + '2'          # 将字符'1'与'2'拼接,结果为'12'
```

上述代码中，加法运算符对于不同类型对象执行不同的操作，这就是多态。在程序中，多态是指基类的同一个方法在不同派生类对象中具有不同的表现和行为，当调用该方法时，程序会根据对象选择合适的方法，如例 11-12 所示。

例 11-12　多态。

```
class Person(object):
    def __init__(self, name):
        self.name = name
    def show(self):
        print('姓名:', self.name)
class Student(Person):
    def __init__(self, name, score):
        super(Student, self).__init__(name)
        self.score = score
    def show(self):
        print('姓名:', self.name, ' 分数:', self.score)
class Staff(Person):
    def __init__(self, name, salary):
        super(Staff, self).__init__(name)
        self.salary = salary
    def show(self):
        print('姓名:', self.name, ' 薪资:', self.salary)
def printInfo(obj):
    obj.show()
s1 = Student('小千', 100)
s2 = Staff('小锋', 10000)
```

```
22 printInfo(s1)
23 printInfo(s2)
```

运行结果如图 11.19 所示。

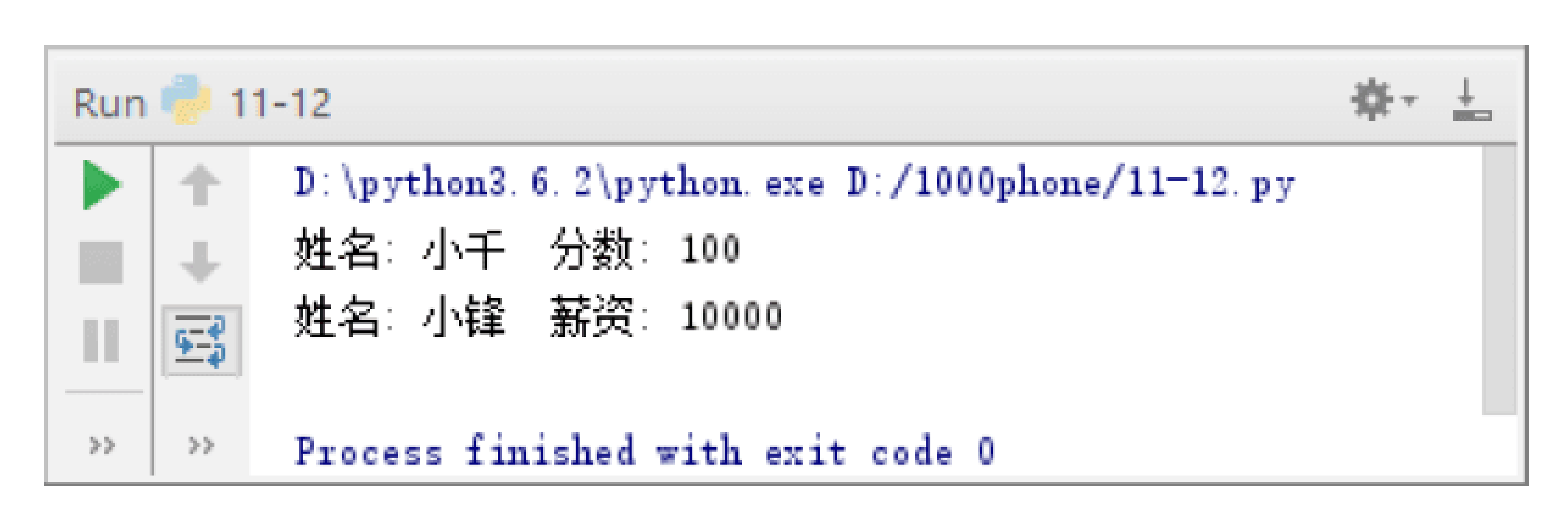


图 11.19 例 11-12 运行结果

在例 11-12 中，Student 与 Staff 都继承自 Person 类，3 个类中都存在 show()。第 22 行与第 23 行通过自定义函数 printInfo 调用各类对象中的 show 属性，程序会根据对象选择调用不同的 show()方法，这种表现形式称为多态。

11.5 设计模式

设计模式描述了软件设计过程中经常碰到的问题及解决方案，它是面向对象设计经验的总结和理论化抽象。通过设计模式，开发者就可以无数次地重用已有的解决方案，无须再重复相同的工作。本节将简单介绍工厂模式与适配器模式。

11.5.1 工厂模式

工厂模式主要用来实例化有共同方法的类，它可以动态决定应该实例化哪一个类，不必事先知道每次要实例化哪一个类。例如在编写一个应用程序时，用户可能会连接各种各样的数据库，但开发者不能预知用户会使用哪个数据库，于是提供一个通用方法，里面包含了各个数据库的连接方案，用户在使用过程中，只需要传入数据库的名字并给出连接所需要的信息即可，如例 11-13 所示。

例 11-13 工厂模式。

```
class Operation(object):
    def connect(self):
        pass
class MySQL(Operation):
    def connect(self):
        print('连接 MySQL 成功')
class SQLite(Operation):
    def connect(self):
```

```
9           print('连接 SQLite 成功')
10  class DB(object):
11      @staticmethod
12      def create(name):
13          name = name.lower()
14          if name == 'mysql':
15              return MySQL()
16          elif name == 'sqlite':
17              return  SQLite()
18          else:
19              print('不支持其他数据库')
20  if __name__ == '__main__':
21      db1 = DB.create('MySQL')
22      db1.connect()
23      db2 = DB.create('SQLite')
24      db2.connect()
```

运行结果如图 11.20 所示。

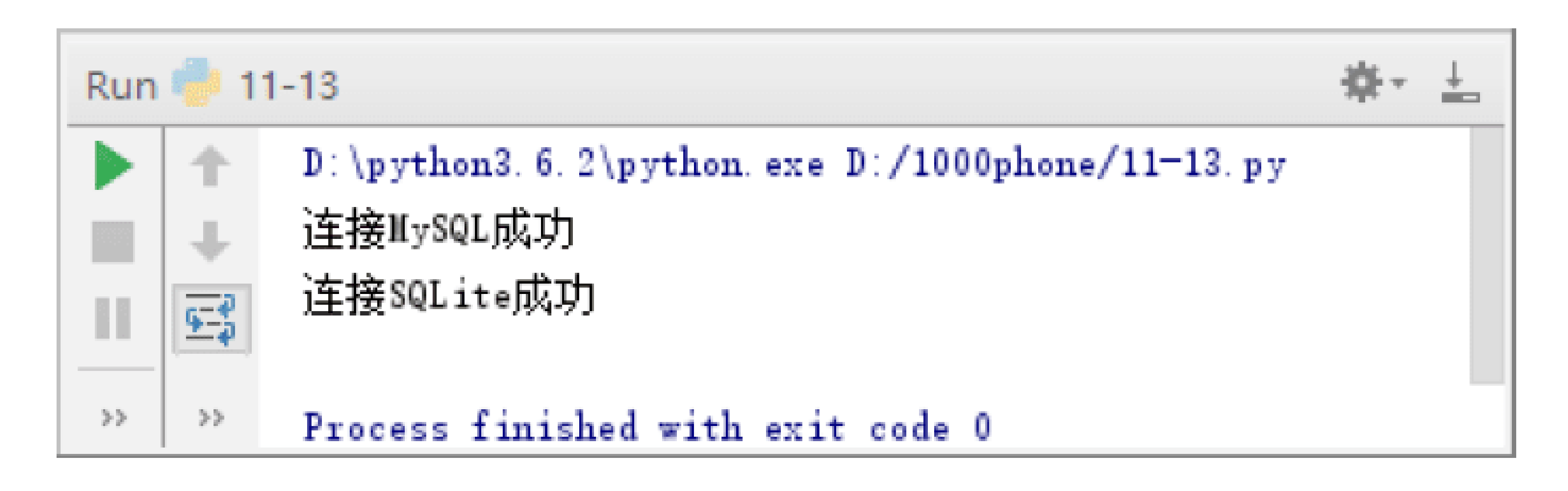


图 11.20　例 11-13 运行结果

在例 11-13 中，第 10 行定义 DB 类，类中定义了一个静态方法 create()，该方法的参数为类名，可以根据类名创建对应的对象，因此称为工厂方法。从程序运行结果可看出，工厂方法可以根据类名创建相应的对象。

11.5.2　适配器模式

适配器模式是指一种接口适配技术，实现两个不兼容接口之间的兼容，例如原程序中存在类 Instrument 与 Person，其中 Instrument 实例对象可以调用 play()方法，Person 实例对象可以调用 act()方法，新程序中增加类 Computer，其实例对象可以调用 execute()方法。现要求类 Instrument 与 Person 的实例对象通过 execute()调用各自的方法，具体如例 11-14 所示。

例 11-14　适配器模式。

```
class Instrument(object): # 乐器类,原程序存在的类
    def __init__(self, name):
        self.name = name
```

```
4      def play(self):
5          print(self.name, '演奏')
6  class Person(object): # 人类,原程序存在的类
7      def   init   (self, name):
8          self.name = name
9      def act(self):
10         print(self.name, '表演')
11 class Computer(object): # 计算机类,新程序添加的类
12     def   init   (self, name):
13         self.name = name
14     def execute(self):
15         print(self.name, '执行程序')
16 class Adapter(object): # 适配器类,用于统一接口
17     def __init__(self, obj, adaptedeMthods):
18         self.obj = obj
19         self.  dict  .update(adaptedeMthods)
20 if __name__ == '__main__':
21     obj1 = Instrument('guitar')
22     Adapter(obj1, dict(execute = obj1.play)).execute()
23     obj2 = Person('xiaoqian')
24     Adapter(obj2, dict(execute = obj2.act)).execute()
```

运行结果如图 11.21 所示。

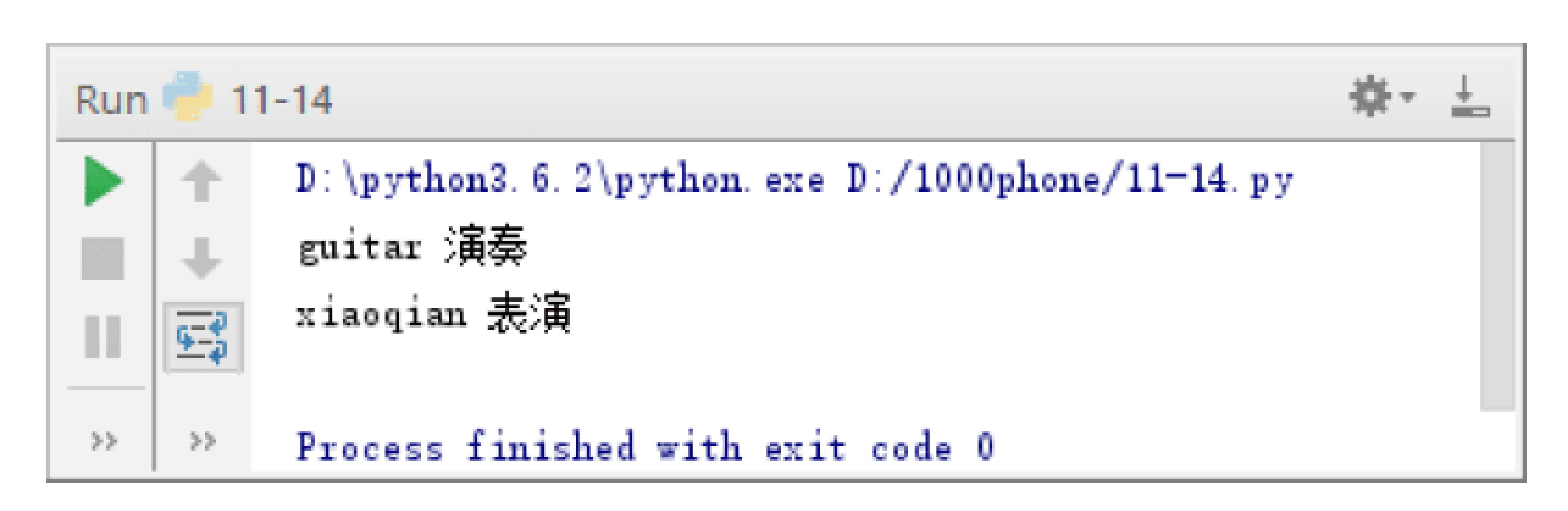


图 11.21　例 11-14 运行结果

在例 11-14 中，第 17 行定义适配器类，第 19 行使用类的内部字典__dict__，它的键是属性名，值是相应属性对象的数据值。第 22、24 行通过适配器类的实例对象实现了新程序与原程序的兼容。

11.6　小　案　例

小伙伴在童年时都看过《猫和老鼠》，本案例实现猫捉老鼠游戏，猫与老鼠的位置用整数代替，游戏开始时，猫与老鼠的位置随机，之后老鼠移动的步数在[−2, −1, 0, 1, 2]中随机选择一个，猫移动的步数通过用户输入，当老鼠位置与猫位置相同时，游戏结束，

具体实现如例 11-15 所示。

例 11-15 猫捉老鼠实现。

```
import random
# 动物类
class Animal(object):
    step = [-2, -1, 0, 1, 2]
    def __init__(self, gm, point = None):
        self.gm = gm
        if point is None:
            self.point = random.randint(0, 50)
        else:
            self.point = point
    def move(self, aStep = random.choice(step)):
        if 0 <= self.point + aStep <= 50:
            self.point += aStep
# 猫类,继承自动物类
class Cat(Animal):
    def __init__(self, gm, point = None):
        super(Cat, self).__init__(gm, point)
        self.gm.setPoint('cat', self.point)
    def move(self):
        aStep = int(input('请输入猫移动的步数：'))
        super(Cat, self).move(aStep)
        self.gm.setPoint('cat', self.point)
 # 老鼠类,继承自动物类
class Mouse(Animal):
    def __init__(self, gm, point = None):
        super(Mouse, self).__init__(gm, point)
        self.gm.setPoint('mouse', self.point)
    def move(self):
        super(Mouse, self).move()
        self.gm.setPoint('mouse', self.point)
# 地图类
class GameMap(object):
    def __init__(self):
        self.catPoint, self.mousePoint = None, None
    # 设置猫或老鼠的位置
    def setPoint(self, obj, point):
        if obj == 'cat':
            self.catPoint = point
        if obj == 'mouse':
            self.mousePoint = point
```

```
    # 判断猫与老鼠是否相遇
    def catched(self):
        print('老鼠:', self.mousePoint, '\t 猫:', self.catPoint)
        if self.mousePoint is not None and self.catPoint is not None \
                and self.mousePoint == self.catPoint:
            return True
# 测试
if __name__ == '__main__':
    gm = GameMap()
    mouse = Mouse(gm)
    cat = Cat(gm)
    while not gm.catched():
        mouse.move()
        cat.move()
    else:
        print('猫抓住老鼠，游戏结束!')
```

运行结果如图 11.22 所示。

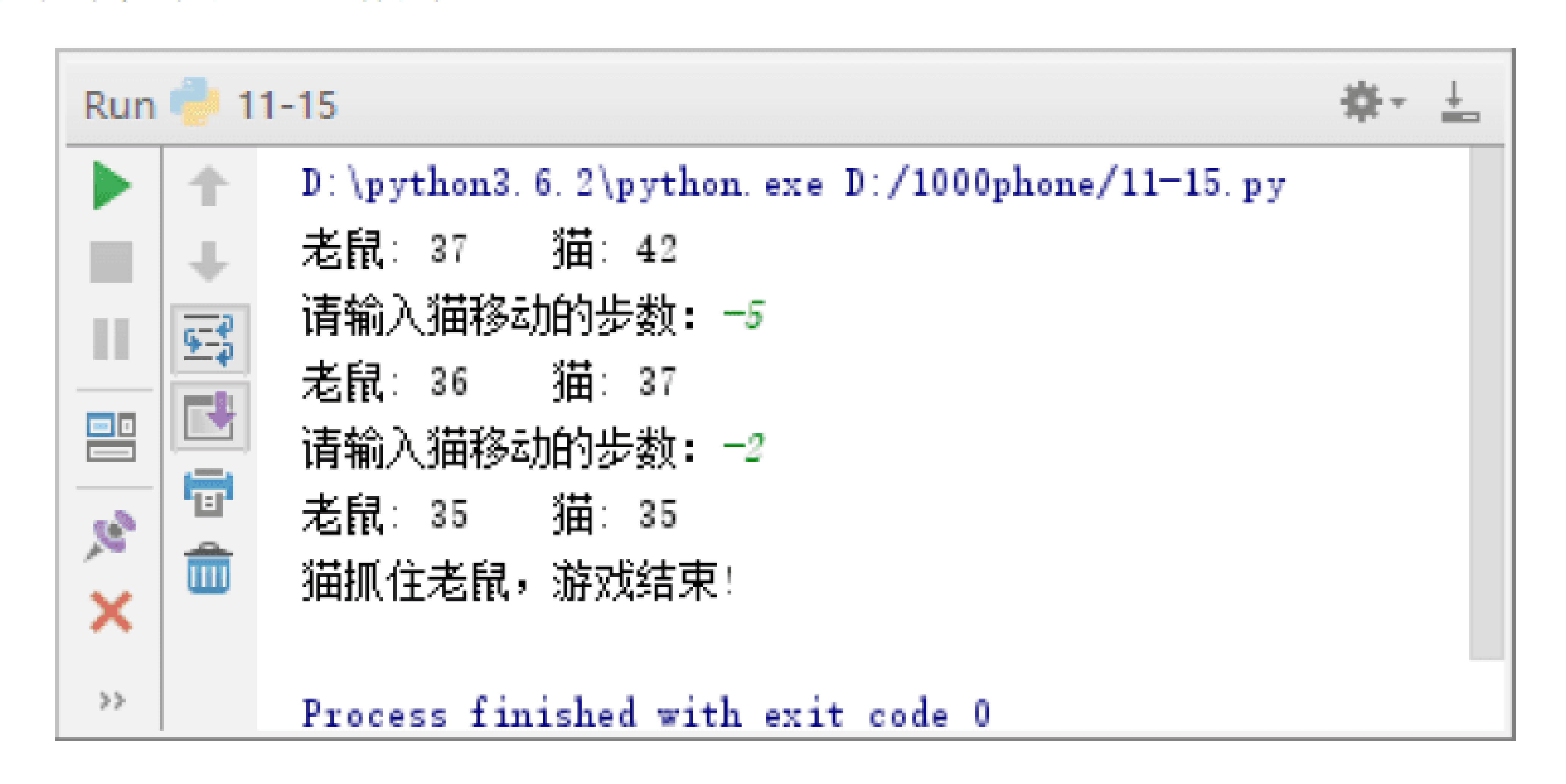


图 11.22 运行结果

在例 11-15 中，定义了 4 个类（Animal、Cat、Mouse、GameMap），其中 Cat 与 Mouse 继承自 Animal 并分别改写了父类中 move()方法。另外，Animal、Cat、Mouse 类中构造方法的第二个参数接受 GameMap 类的实例对象。

11.7 本 章 小 结

本章主要介绍了面向对象的三大特征，包括封装、继承与多态。在掌握面向对象编程后，还需了解设计模式，它为编写大型应用程序提供了思路。学习完本章内容，应加深对面向对象程序设计的理解，并能够将该方法运用到实际开发中。

11.8 习　　题

1．填空题

（1）________是指将对象的属性和行为封装起来。
（2）________是面向对象程序设计提高重用性的重要措施。
（3）________是指一种行为对应着多种不同的实现。
（4）Python 中所有的类都继承自________类。
（5）继承分为单一继承与________。

2．选择题

（1）Python 中通过在属性名前加（　　）个下画线来表明私有属性。
A．0　　B．1　　C．2　　D．4
（2）下列选项中，与 class A 等价的写法是（　　）。
A．class A:(object)　　B．class A:Object
C．class AObject　　D．class A(object)
（3）下列选项中，关于多重继承正确的是（　　）。
A．class A:B, C　　B．class A:(B, C)
C．class A(B, C):　　D．class A(B:C):
（4）派生类通过（　　）可以调用基类的构造方法。
A．__init__()　　B．super()
C．__del__()　　D．派生类名
（5）若基类与派生类中有同名实例方法，则通过派生类实例对象调用（　　）中方法。
A．基类　　B．派生类
C．先基类后派生类　　D．先派生类后基类

3．思考题

（1）简述面向对象的三大特征。
（2）什么是工厂模式？

4．编程题



编写程序，定义动物 Animal 类，由其派生出猫类（Cat）和狮子类（Lion），二者都包含 sound()实例方法，要求根据派生类对象的不同调用各自的 sound()方法。

第 12 章 chapter 12

文　件

本章学习目标

- 理解文件的概念。
- 掌握文件的操作。
- 掌握目录的操作。

程序在运行时将数据加载到内存中，内存中的数据是不能永久保存的，这时就需要将数据存储起来以便后期多次使用。通常是将数据存储在文件或数据库中，而数据库最终还是要以文件的形式存储到介质上，因此掌握文件处理是十分有必要的。

12.1 文件概述

相信大家对文件并不陌生，它可以存储文字、图片、音乐、视频等，如图 12.1 所示。总之，文件是数据的集合，可以有不同的类型。

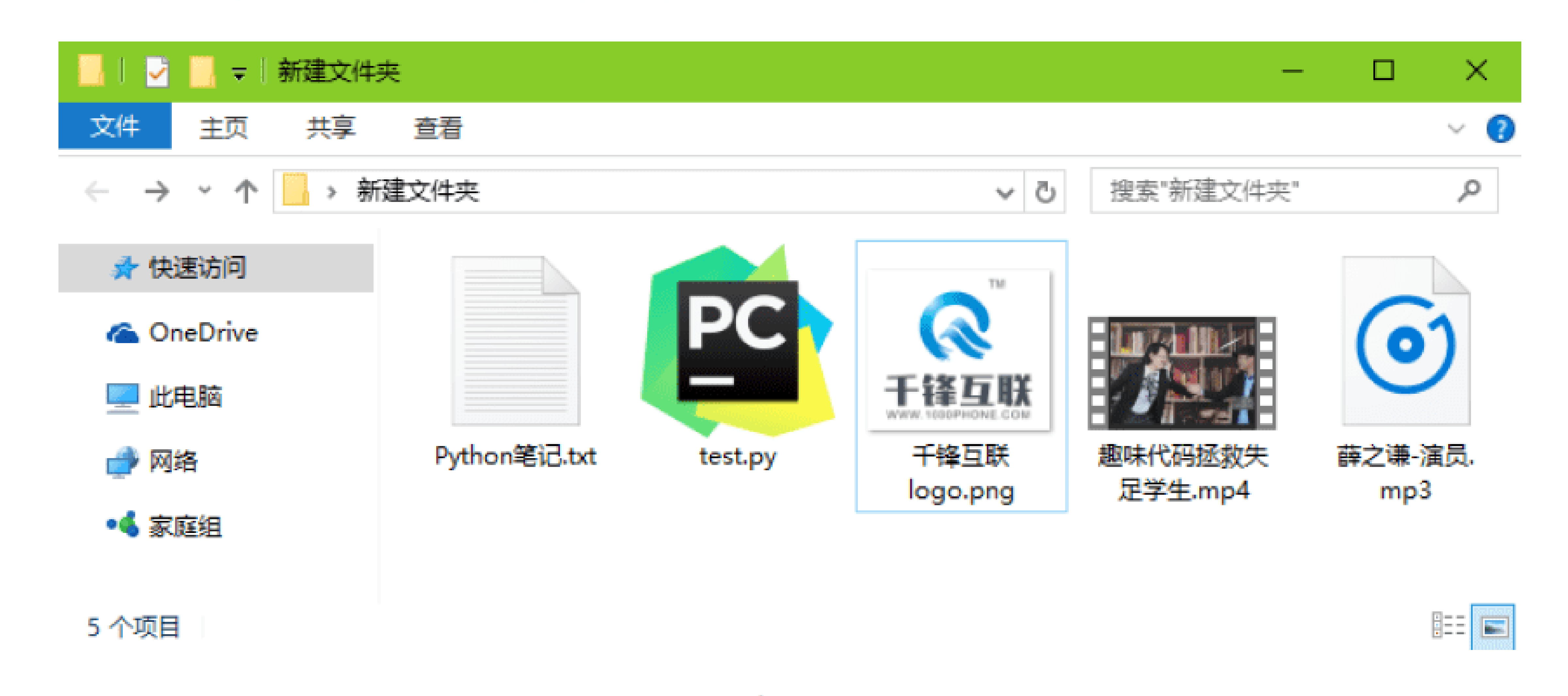


图 12.1　各种不同类型的文件

按数据的组织形式，文件大致可以分为如下两类。

1. 文本文件

文本文件是一种由若干字符构成的文件，可以用文本编辑器进行阅读或编辑。以 txt、

py、html 等为后缀的文件都是文本文件。

2. 二进制文件

二进制文件一般是指不能用文本编辑器阅读或编辑的文件。以 mp3、mp4、png 等为后缀的文件都是二进制文件，如果想要打开或修改这些文件，必须通过特定软件进行，比如用 Photoshop 软件可以编辑图像文件。

从本质上讲，文本文件也是二进制文件，因为计算机处理的全是二进制数据。

12.2 文件操作

通过程序操作文件与手动操作文件类似，通常需要经过 3 个步骤：打开文件、读或写数据、关闭文件。

12.2.1 打开文件

对文件所有的操作都是在打开文件之后进行的，打开文件使用 open()函数来实现，其语法格式如下：

```
open(file[, mode = 'r' [,…]])
```

该函数返回一个文件对象，通过它可以对文件进行各种操作，参数列表中参数的说明如表 12.1 所示。

表 12.1 open()函数的各参数说明

参数	说明
file	被打开的文件名
mode	文件打开模式，默认是只读模式

例如打开文件名为 test.txt 的文件，具体示例如下：

```
f1 = open('test.txt')                  # 打开当前目录下的 test.txt 文件
f2 = open('../test.txt')               # 打开上级目录下的 test.txt 文件
f3 = open('D:/1000phone/test.txt')  # 打开 D:/1000phone 目录下的 test.txt 文件
```

示例中使用 open()函数打开文件时使用只读模式打开，此时必须保证文件是存在的，否则会报文件不存在的错误。

Python 中打开文件的模式有多种，具体如表 12.2 所示。

在表 12.2 中，'r'表示从文件中读取数据，'w'表示向文件中写入数据，'a'表示向文件中追加数据，'+'可以与以上 3 种模式（'r'、'w'、'a'）配合使用，表示同时允许读和写。另外，当需要处理二进制文件时，则需要提供'b'给 mode 参数，例如'rb'用于读取二进制文件。

表 12.2　文件打开模式

mode	权　限			读/写格式	删除原内容	文件不存在	文件指针初始位置
	读	写	追加				
'r'	√			文本		产生异常	文件开头
'r+'	√	√		文本		产生异常	文件开头
'rb+'	√	√		二进制		产生异常	文件开头
'w '		√		文本	√	新建文件	文件开头
'w+'	√	√		文本	√	新建文件	文件开头
'wb+'	√	√		二进制	√	新建文件	文件开头
'a'			√	文本		新建文件	文件末尾
'a+'	√	√	√	文本		新建文件	文件末尾
'ab+'	√	√	√	二进制		新建文件	文件末尾

12.2.2　关闭文件

当对文件内容操作完以后，一定要关闭文件，这样才能保证所修改的数据保存到文件中，同时也可以释放内存资源供其他程序使用。关闭文件的语法格式如下：

```
文件对象名.close()
```

接下来演示文件的打开与关闭，如例 12-1 所示。

例 12-1　文件的打开与关闭。

```
f = open('test.txt', 'a+')   # 以追加模式读写 test.txt
f.close()                    # 关闭文件
```

运行结果如图 12.2 所示。

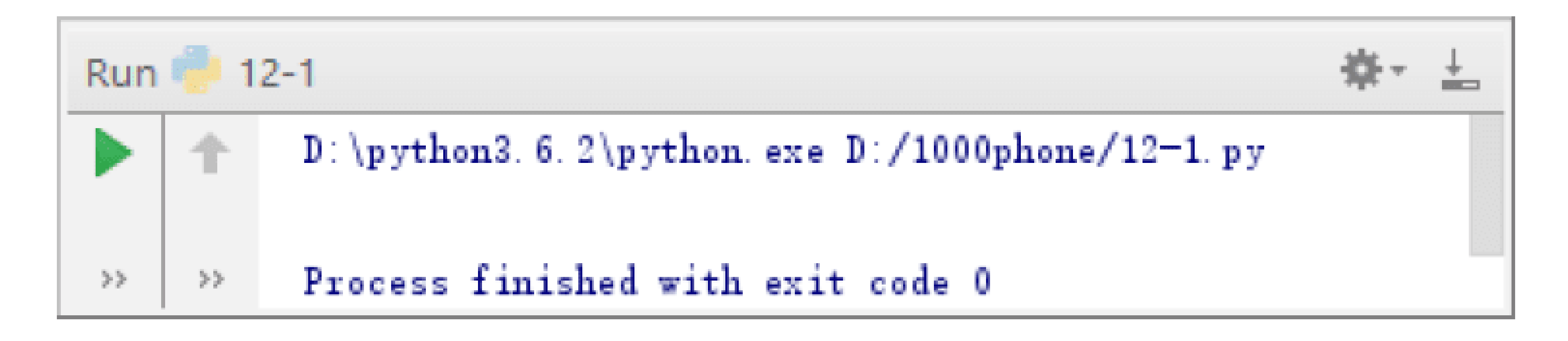


图 12.2　例 12-1 运行结果

在例 12-1 中，通过 open()函数打开文件 test.txt，此时返回一个文件对象并赋值给 f，最后通过文件对象调用 close()方法关闭文件。

此处需注意，即使使用了 close()方法，也无法保证文件一定能够正常关闭。例如，在打开文件之后和关闭文件之前发生了错误导致程序崩溃，这时文件就无法正常关闭。因此，在管理文件对象时推荐使用 with 关键字，可以有效地避免这个问题，具体示例如下：

```
with open('test.txt', 'r+') as f:
```

```
    # 通过文件对象 f 进行读写操作
```

使用 with-as 语句后，就不需要再显式使用 close()方法。另外 with-as 语句还可以打开多个文件，具体示例如下：

```
with open('test1.txt', 'r+') as f1, open('test2.txt', 'a+') as f2:
    # 通过文件对象 f1、f2 分别操作 test1.txt、test2.txt 文件
```

从上述示例可看出，with-as 语句极大地简化了文件打开与关闭操作，这对保持代码的优雅性有极大的帮助。

12.2.3 读文本文件

打开文件成功后将返回一个文件对象，对文件内容的读取可以通过该对象来实现，该对象有 3 种方法可以获取文件内容，具体如下所示：

1. read()方法

read()方法可以从文件中读取内容，其语法格式如下：

```
文件对象.read([size])
```

该方法表示从文件中读取 size 个字节或字符作为结果返回，如果省略 size，则表示读取所有内容，如例 12-2 所示。

例 12-2 read()方法的使用。

```
with open('test.txt') as f:
    str1 = f.read(4)      # 读取 4 个字符
    str2 = f.read()       # 读取剩余所有字符
    print(str1, str2, sep = '\n')
```

假设 test.txt 文件内容如图 12.3 所示。

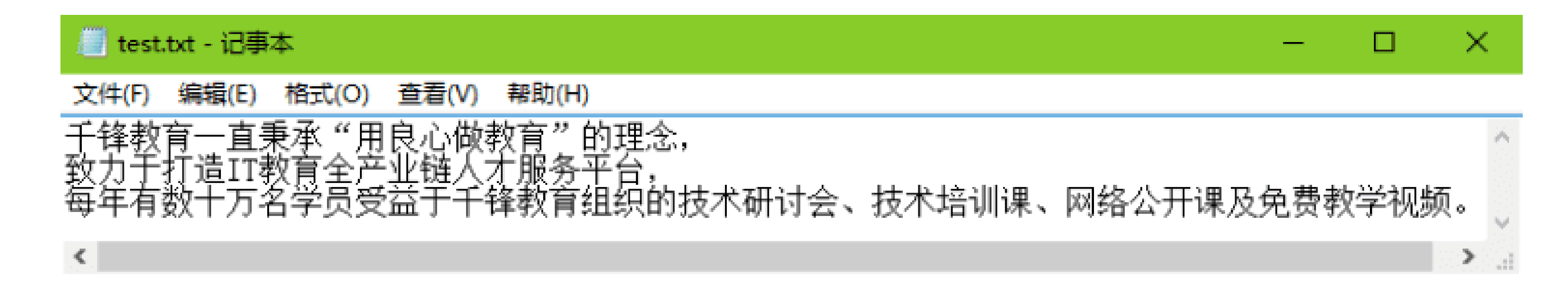


图 12.3 test.txt 文件内容

程序运行结果如图 12.4 所示。

在例 12-2 中，程序使用 with-as 语句打开 test.txt 文件，先读取 4 个字符组成字符串（'千锋教育'）并赋给 str1，接着再读取剩余的字符串赋给 str2。

2. readlines()方法

readlines()方法可以读取文件中的所有行，其语法格式如下：

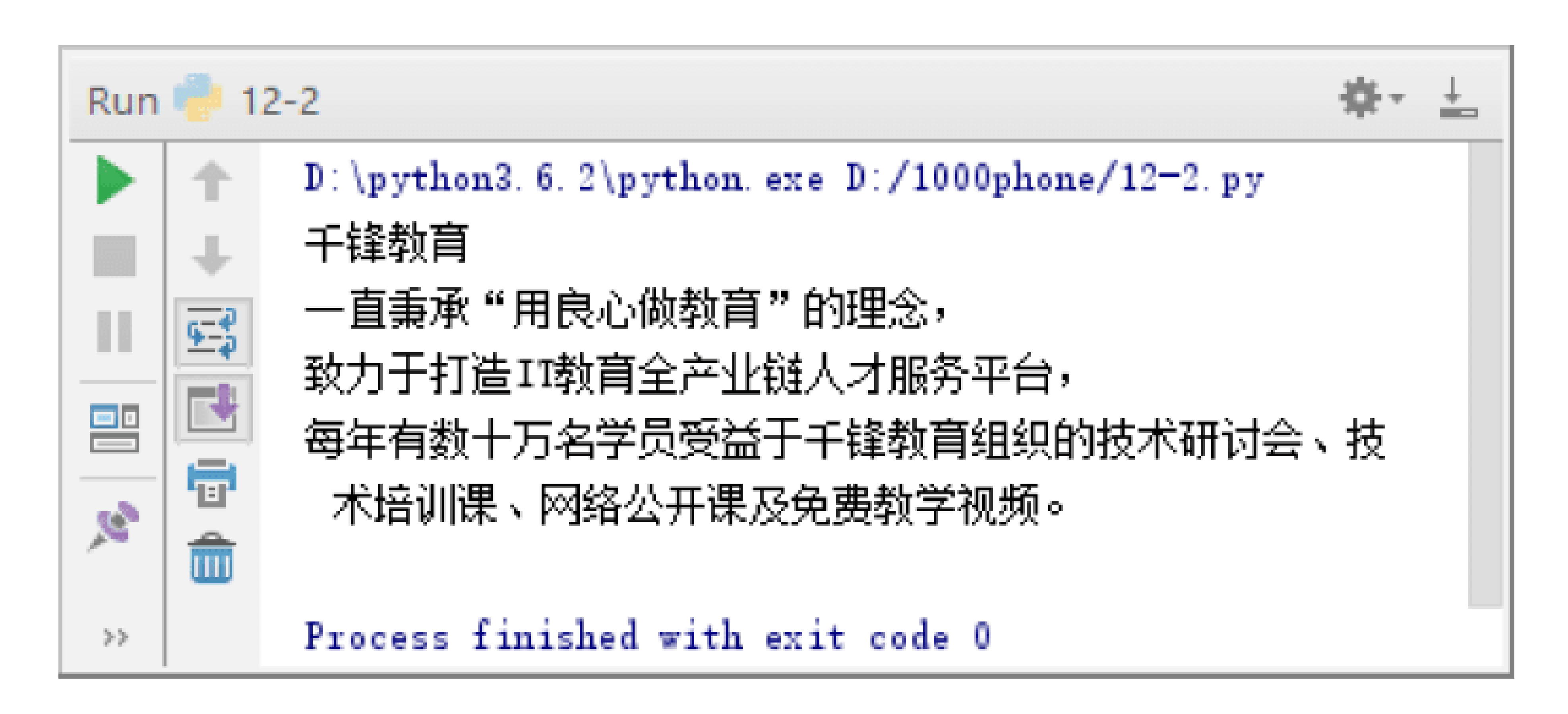


图 12.4　例 12-2 运行结果

```
文件对象.readlines()
```

该方法将文件中的每行内容作为一个字符串存入列表中并返回该列表，如例 12-3 所示。

例 12-3　readlines()方法的使用。

```
1  with open('test.txt') as f:
2      str = f.readlines()  # 读取所有行内容
3      print(str)
```

运行结果如图 12.5 所示。

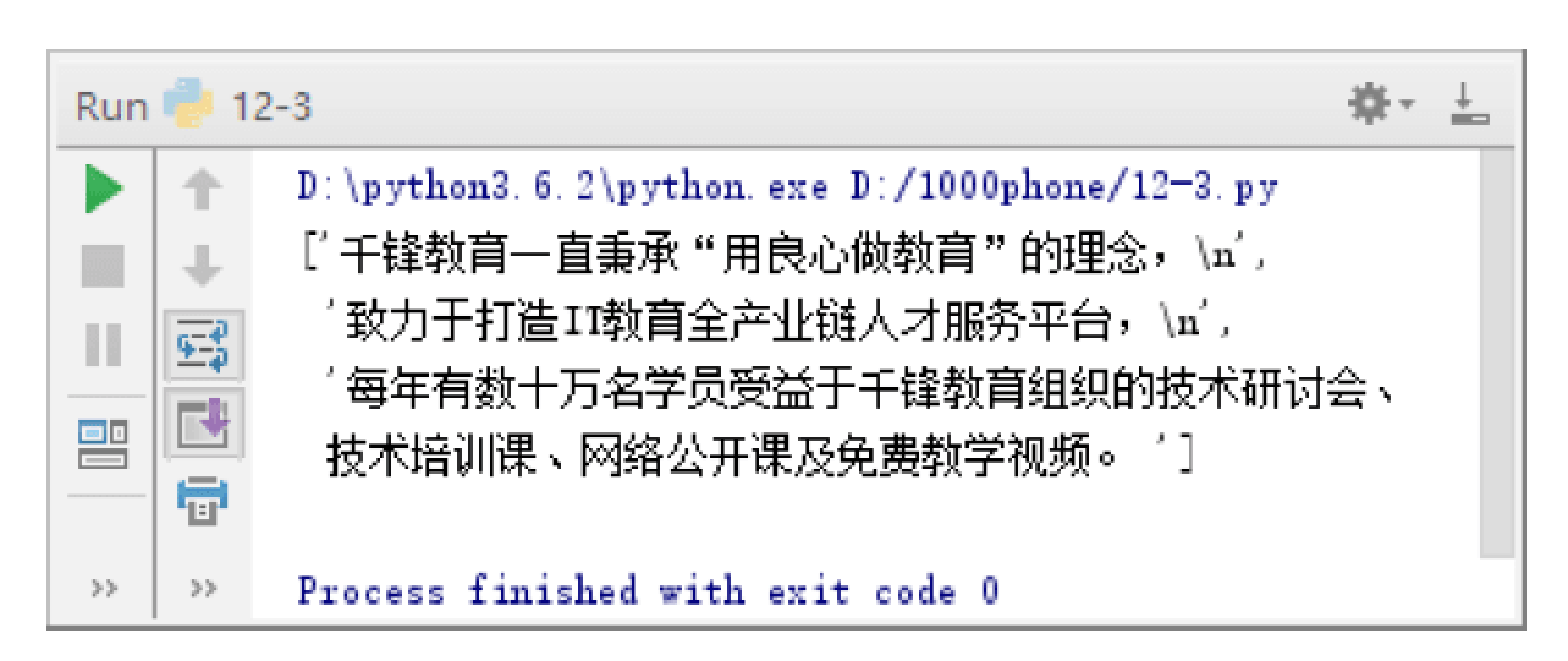


图 12.5　例 12-3 运行结果

在例 12-3 中，第 2 行使用 readlines()方法读取文件 test.txt 中每行内容并存入列表中。此处需注意，readlines()方法一次性读取文件中的所有行，如果文件非常大，使用 readlines()方法就会占用大量的内存空间，读取的过程也较长，因此不建议对大文件使用该方法。

3．readline()方法

readline()方法可以逐行读取文件的内容，其语法格式如下：

```
文件对象.readline()
```

该方法将从文件中读取一行内容作为结果返回，如例 12-4 所示。

例 12-4　readline()方法的使用。

```
with open('test.txt') as f:
    while True:
        str = f.readline()    # 读取一行内容
        if not str:           # 若没读取到内容,则退出循环
            break
        print(str, end = '') # 若读取到内容,则打印内容
```

运行结果如图 12.6 所示。

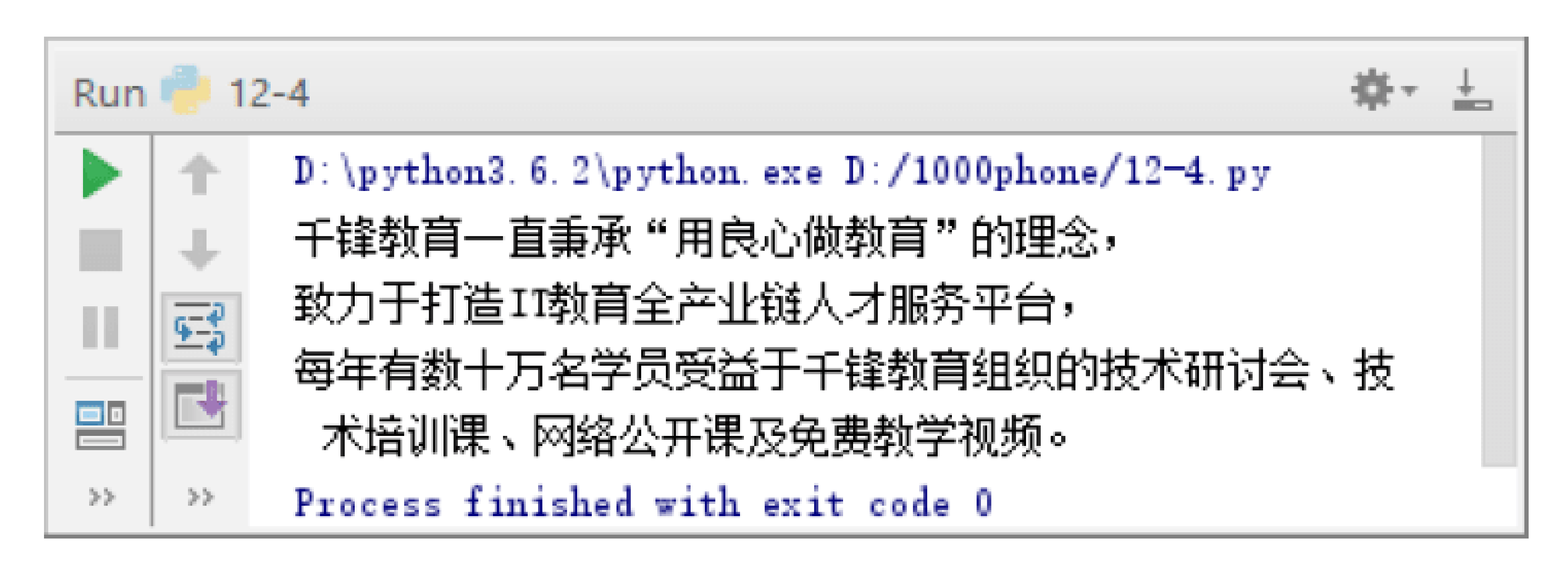


图 12.6 例 12-4 运行结果

在例 12-4 中，程序通过 while 循环每次从文件中读取一行，当未读取到内容时，退出循环。

4. in 关键字

除了上述几种方法外，还可以通过 in 关键字读取文件，如例 12-5 所示。

例 12-5 in 关键字的使用。

```
with open('test.txt') as f:
    for line in f:
        print(line, end = '')
```

运行结果如图 12.7 所示。

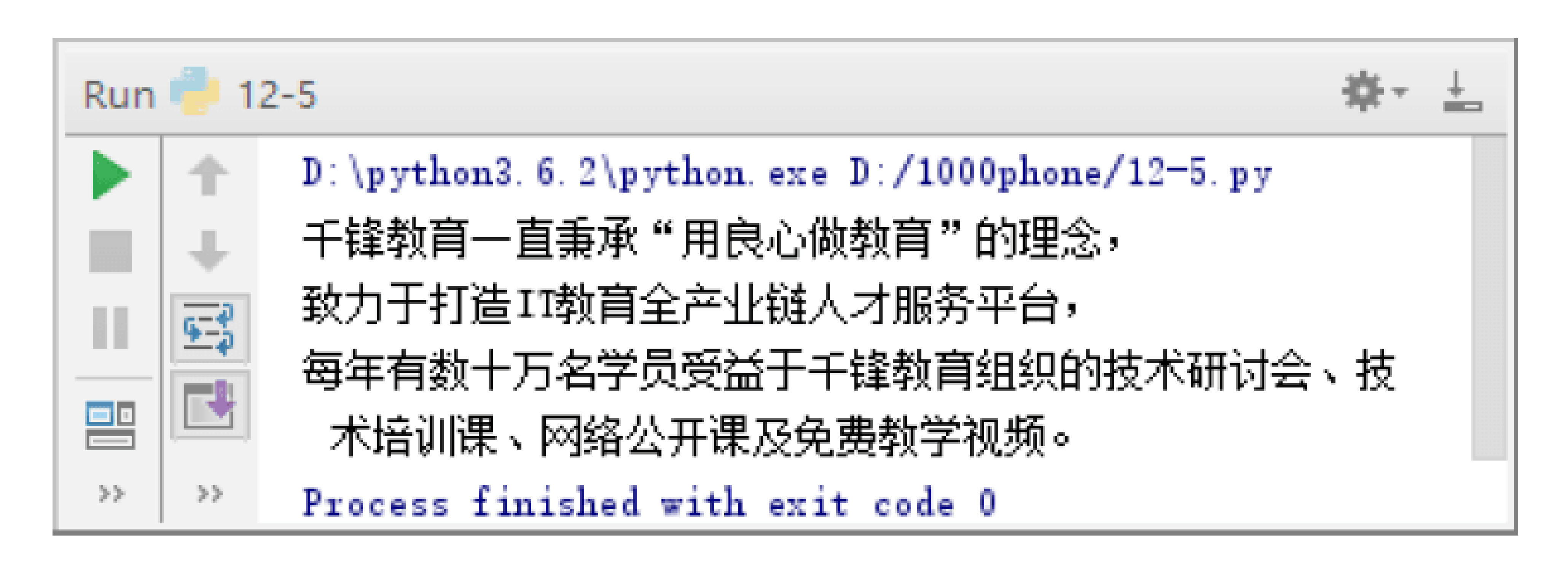


图 12.7 例 12-5 运行结果

在例 12-5 中，程序通过 for 循环每次从文件中读取一行，当未读取到内容时，退出循环。

12.2.4 写文本文件

文件中写入内容也是通过文件对象来完成，可以使用 write()方法或 writelines()方法来实现。

1．write()方法

write()方法可以实现向文件中写入内容，其语法格式如下：

```
文件对象.write(s)
```

该方法表示将字符串 s 写入文件中，如例 12-6 所示。

例 12-6 write()方法的使用。

```
with open('test.txt', 'w') as f:
    f.write('扣丁学堂\n')
```

程序运行结束后，在程序文件所在路径下打开 test.txt 文件，其内容如图 12.8 所示。

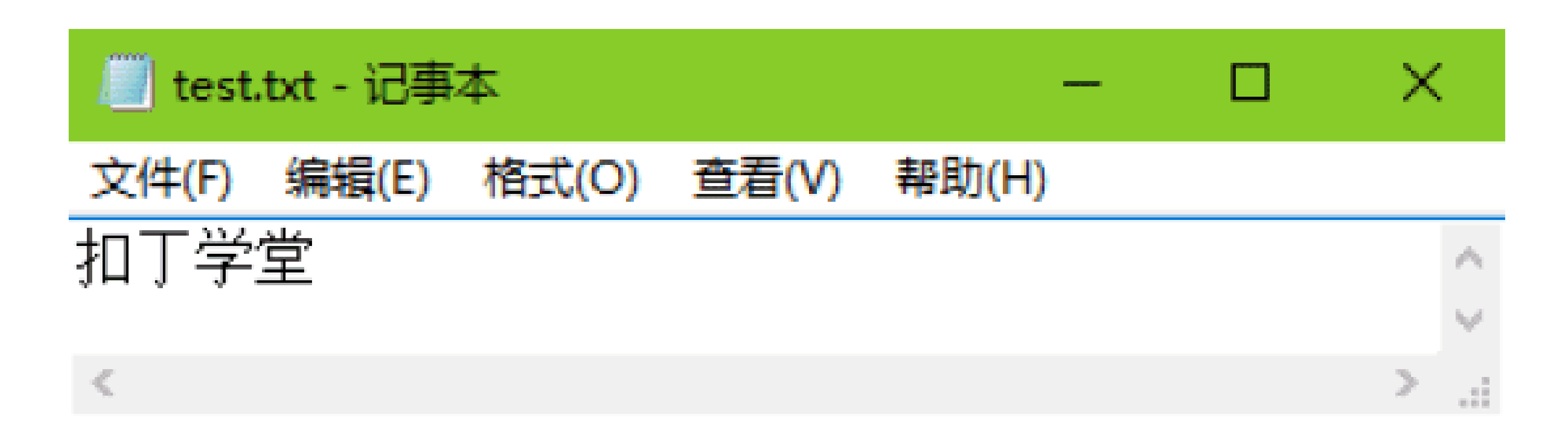


图 12.8 例 12-6 运行结果

在例 12-6 中，程序通过 write()方法向 test.txt 文件中写入'扣丁学堂\n'。注意如果 test.txt 文件在打开之前存在，则先清空文件内容，再写入'扣丁学堂\n'。

2．writelines()方法

writelines()方法向文件中写入字符串列表，其语法格式如下：

```
文件对象.writelines(s)
```

该方法将列表 s 中的每个字符串元素写入文件中，如例 12-7 所示。

例 12-7 writelines()方法的使用。

```
s = ['千锋教育', '扣丁学堂', '好程序员特训营']
with open('test.txt', 'w') as f:
    f.writelines(s)
```

程序运行结束后，在程序文件所在路径下打开 test.txt 文件，其内容如图 12.9 所示。

在例 12-7 中，程序通过 writelines ()方法将列表 s 中的元素写入 test.txt 文件，注意写入的字符串之间没有换行。

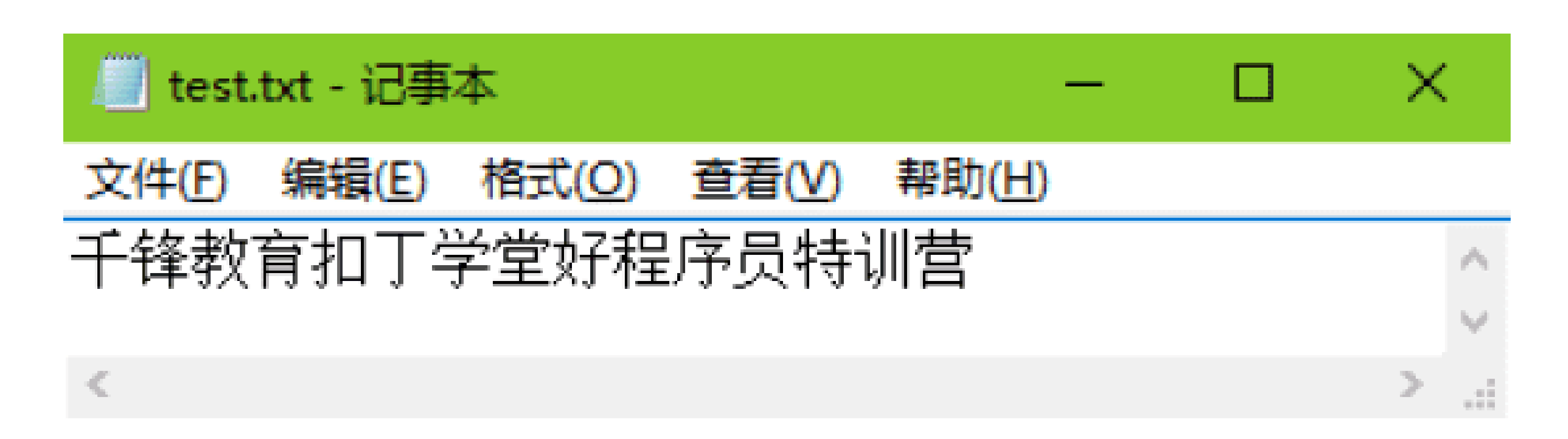


图 12.9　例 12-7 运行结果

12.2.5　读写二进制文件

文本文件使用字符序列来存储数据，而二进制文件使用字节序列存储数据，它只能被特定的读取器读取。Python 中的 pickle 模块可以将数据序列化。

序列化是指将对象转化成一系列字节存储到文件中，而反序列化是指程序从文件中读取信息并用来重构上一次保存的对象。

pickle 模块中的 dump()函数可以实现序列化操作，其语法格式如下：

```
dump(obj, file, [,protocol = 0])
```

该函数表示将对象 obj 保存到文件 file 中，参数 protocol 是序列化模式，默认值为 0，表示以文本的形式序列化，protocol 的值还可以是 1 或 2，表示以二进制的形式序列化。

pickle 模块中的 load ()函数可以实现反序列化操作，其语法格式如下：

```
load(file)
```

该函数表示从文件 file 中读取一个字符串，并将它重构为原来的 python 对象。

接下来演示使用 pickle 模块实现序列化和反序列化操作，如例 12-8 所示。

例 12-8　使用 pickle 模块实现序列化和反序列化操作。

```
1  import pickle                      # 导入 pickle 模块
2  data1 = {'小千': [18, '女', 100],
3           '小锋': [19, '男', 98.5],
4           '小扣': [18, '男', 60] }
5  data2 = ['千锋教育', '扣丁学堂', '好程序特训营']
6  with open('test.dat', 'wb') as f1:
7      pickle.dump(data1, f1)         # 将字典序列化
8      pickle.dump(data2, f1, 1)      # 将列表序列化
9  with open('test.dat', 'rb') as f2:
10     data3 = pickle.load(f2)        # 重构字典
11     data4 = pickle.load(f2)        # 重构列表
12 print(data3, data4)
```

运行结果如图 12.10 所示。

在例 12-8 中，第 7 行和第 8 行通过 dump()函数将 data1 与 data2 进行序列化后写入 test.dat 文件中，第 10 行和第 11 行通过 load()函数从文件 test.dat 中读取数据并进行反序列化操作。

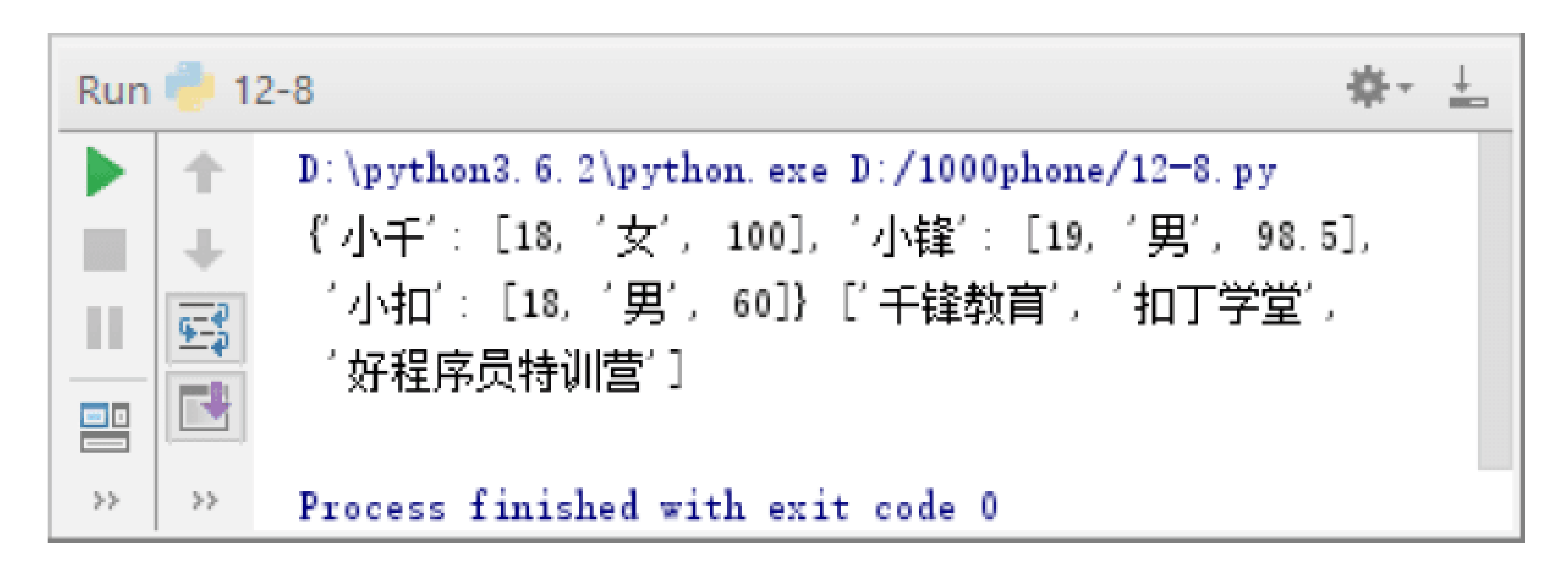


图 12.10　例 12-8 运行结果

12.2.6　定位读写位置

文件指针是指向一个文件的指针变量，用于标识当前读写文件的位置，通过文件指针就可对它所指的文件进行各种操作。

tell()方法可以获取文件指针的位置，其语法格式如下：

```
文件对象.tell()
```

该方法返回一个整数，表示文件指针的位置，如例 12-9 所示。

例 12-9　tell()方法的使用。

```
1  with open('test.txt', 'w+') as f:
2      n = f.tell()      # 文件指针指向文件头,值为 0
3      print(n)
4      f.write('www.qfedu.com')
5      n = f.tell()      # 文件指针指向文件尾
6      print(n)
```

运行结果如图 12.11 所示。

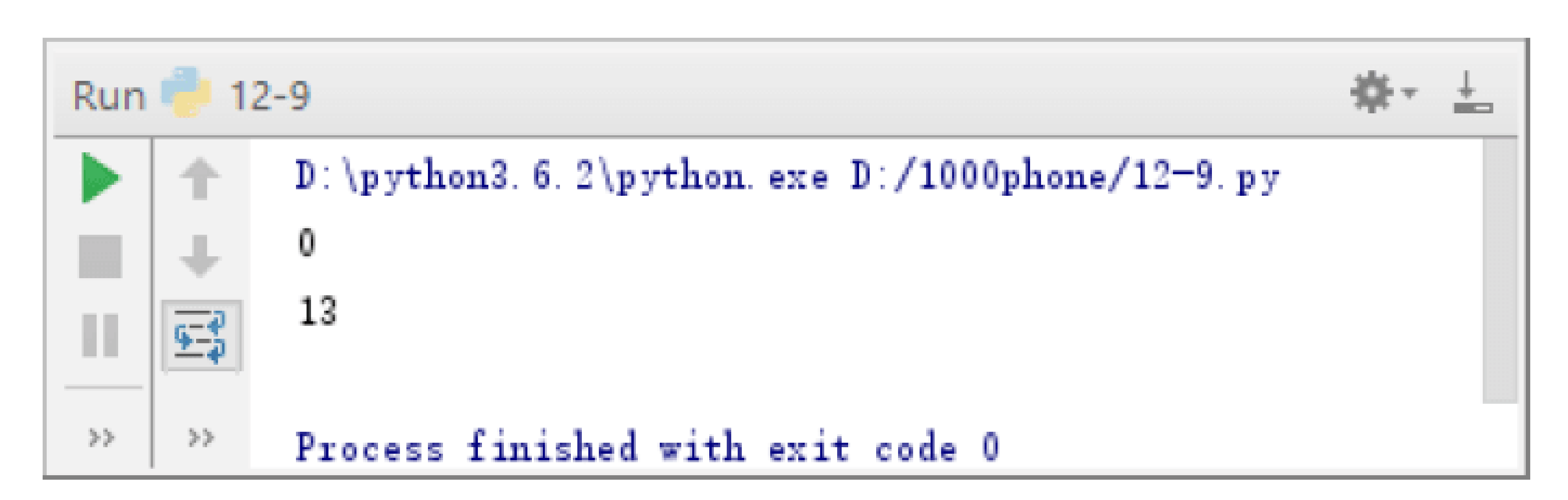


图 12.11　例 12-9 运行结果

在例 12-9 中，第 2 行获取文件指针位置为 0，表示处于文件头，第 4 行向文件中写入字符串'www.qfedu.com'，长度为 13，此时文件指针处于文件尾。

seek()方法可以移动文件指针位置，其语法格式如下：

```
文件对象.seek((offset[, where = 0]))
```

其中，参数 offset 表示移动的偏移量，单位为字节，其值为正数时，文件指针向文件尾

方向移动；其值为负数时，文件指针向文件头方向移动。参数 where 指定从何处开始移动，其值可以为 0、1、2，具体含义如下所示：

- 0——表示文件头。
- 1——表示当前位置。
- 2——表示文件尾。

接下来演示 seek()方法的使用，如例 12-10 所示。

例 12-10 seek()方法的使用。

```
with open('test.txt', 'w+') as f:
    print(f.tell())  # 文件指针处于文件头
    f.write('qfedu/com')
    print(f.tell())  # 文件指针处于文件尾
    f.seek(5,0)      # 文件指针处于位置 5
    print(f.tell())
    f.write('.')
    print(f.tell())
```

运行结果如图 12.12 所示。

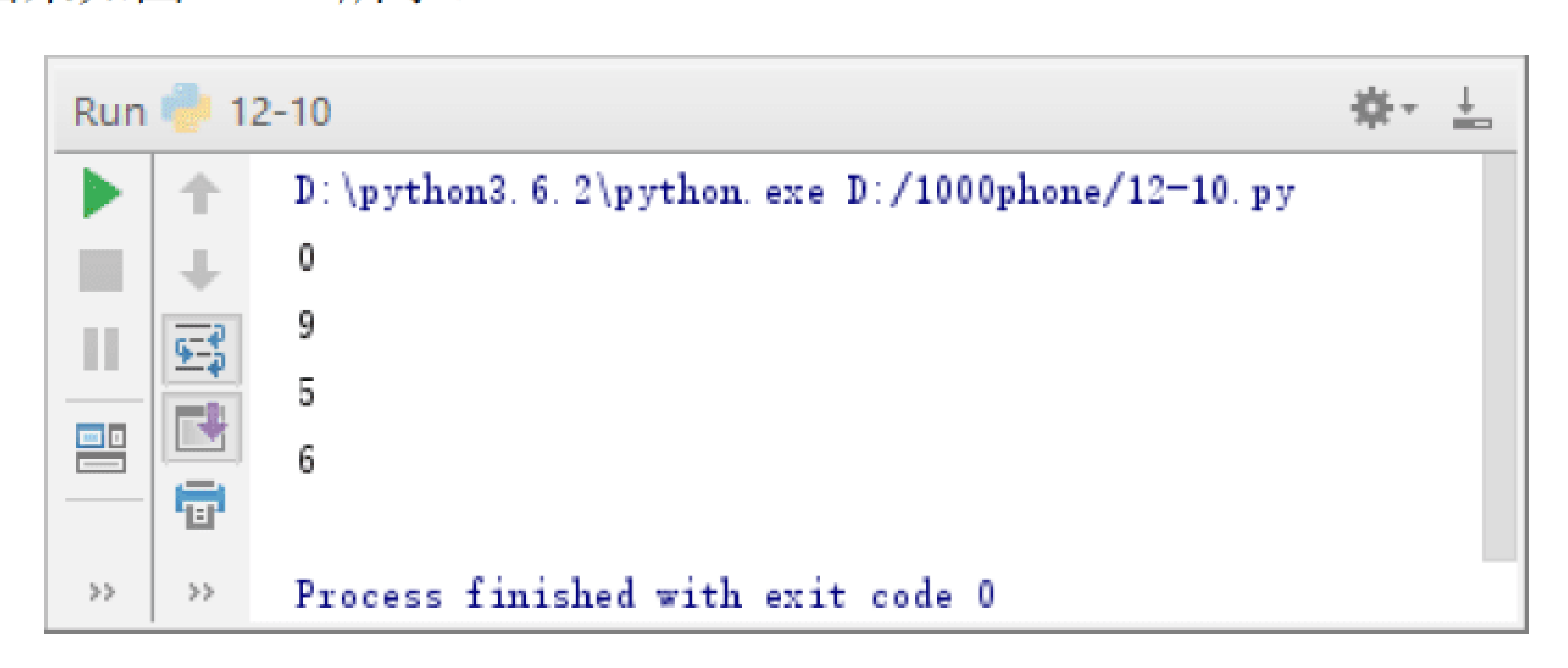


图 12.12 例 12-10 运行结果

程序运行结束后， test.txt 文件内容如图 12.13 所示。

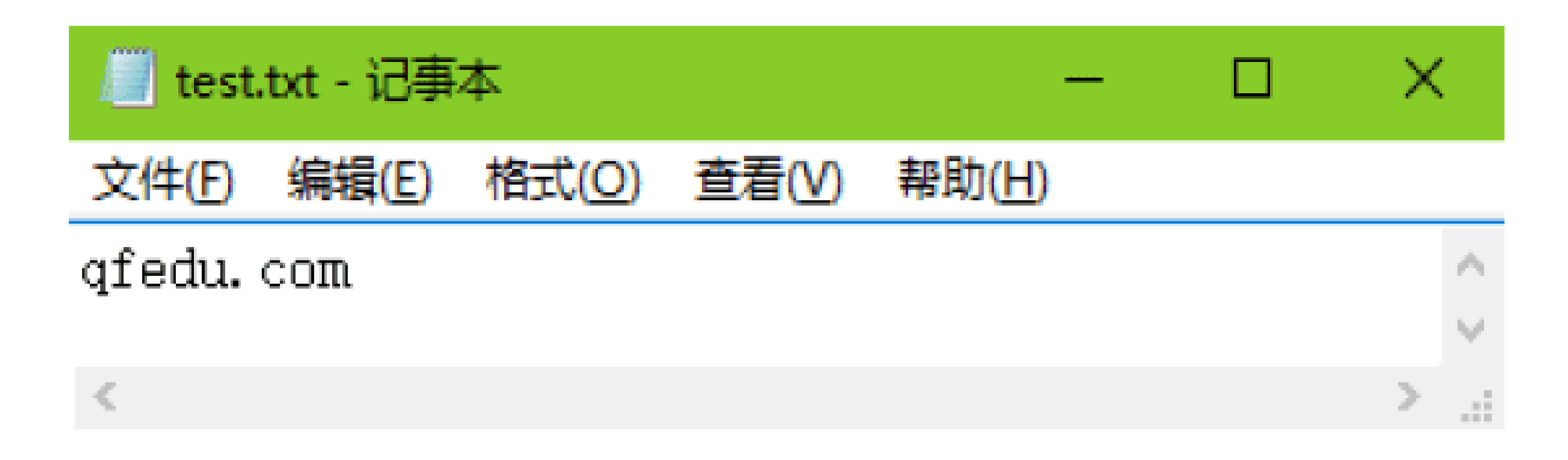


图 12.13 test.txt 文件内容

在例 12-10 中，程序通过移动文件指针将“/”替换为“.”。

12.2.7 复制文件

在日常生活中，经常需要将文件从一个路径下复制到另一个路径下。在 Python 中，

shutil 模块的 copy()函数可以实现复制文件，其语法格式如下：

```
shutil.copy(src, dst)
```

该函数表示将文件 src 复制为 dst，如例 12-11 所示。

例 12-11 shutil 模块的 copy()函数的使用。

```
import shutil # 导入 shutil 模块
shutil.copy('D:/1000phone/test.txt', 'copytest.txt')
```

程序运行结束后，在目录“D:/1000phone/”下会生成一个 copytest.txt 文件。

12.2.8 移动文件

在日常生活中，经常需要将文件从一个路径下移动到另一个路径下。在 Python 中，shutil 模块的 move ()函数可以实现移动文件，其语法格式如下：

```
shutil.move(src, dst)
```

该函数表示将文件 src 移动到 dst，如例 12-12 所示。

例 12-12 shutil 模块的 move ()函数的使用。

```
import shutil # 导入 shutil 模块
shutil.move('D:/1000phone/copytest.txt', '../copytest.txt')
```

程序运行结束后，文件 copytest.txt 从目录“D:/1000phone/”移动到目录“D:/”。

12.2.9 重命名文件

在 Python 中，os 模块的 rename()函数可以重命名文件，其语法格式如下：

```
os.rename(src, dst)
```

该函数表示将 src 重命名为 dst，如例 12-13 所示。

例 12-13 os 模块的 rename()函数的使用。

```
import os # 导入 os 模块
os.rename('D:/copytest.txt', 'D:/copytest1.txt')
```

程序运行结束后，文件 copytest.txt 被重命名为 copytest1.txt。

12.2.10 删除文件

在 Python 中，os 模块的 remove ()函数可以删除文件，其语法格式如下：

```
os.remove(src)
```

该函数表示将文件 src 删除，如例 12-14 所示。

例 12-14 os 模块的 remove ()函数的使用。

```
import os # 导入 os 模块
os.remove('D:/copytest1.txt')
```

程序运行结束后，文件 copytest1.txt 被删除。

12.3 目录操作

在开发中，随着文件数量的增多，就需要创建目录来管理文件，本节讲解有关文件目录的操作，该操作需要导入 os 模块。

12.3.1 创建目录

os 模块的 mkdir()函数可以创建目录，其语法格式如下：

```
os.mkdir(path)
```

参数 path 指定要创建的目录，如例 12-15 所示。

例 12-15 os 模块的 mkdir()函数的使用。

```
import os # 导入 os 模块
os.mkdir('D:/1000phone/codingke')
```

程序运行结束后，在目录 D:/1000phone/下创建出一个目录 codingke。此处需注意，该函数只能创建一级目录；如果需要创建多级目录，则可以使用 makedirs()函数，其语法格式如下：

```
os.makedirs(path1/path2…)
```

参数 path1 与 path2 形成多级目录，具体示例如下：

```
import os # 导入 os 模块
os.makedirs('D:/1000phone/goodprogrammer/test')
```

程序运行结束后，目录结构为 D:/1000phone/goodprogrammer/test。

12.3.2 获取目录

os 模块的 getcwd()函数可以获取当前目录，其语法格式如下：

```
os.getcwd()
```

该函数的使用比较简单，如例 12-16 所示。

例 12-16 os 模块的 getcwd()函数的使用。

```
import os # 导入 os 模块
res = os.getcwd()
print(res)
```

运行结果如图 12.14 所示。

```
Run 12-16
D:\python3.6.2\python.exe D:/1000phone/12-16.py
D:\1000phone

Process finished with exit code 0
```

图 12.14 例 12-16 运行结果

从程序运行结果可以看出，本程序的文件在 D:\1000phone 目录中。

另外，os 模块的 listdir()函数可以获取指定目录中包含的文件名与目录名，其语法格式如下：

```
os.listdir(path)
```

其中，参数 path 指定要获取目录的路径，如例 12-17 所示。

例 12-17 os 模块的 listdir()函数的使用。

```
import os # 导入 os 模块
res = os.listdir('D:/1000phone')
print(res)
```

运行结果如图 12.15 所示。

```
Run 12-17
D:\python3.6.2\python.exe D:/1000phone/12-17.py
['.idea', '12-1.py', '12-10.py', '12-11.py',
 '12-12.py', '12-13.py', '12-14.py', '12-15.py',
 '12-16.py', '12-17.py', '12-2.py', '12-3.py',
 '12-4.py', '12-5.py', '12-6.py', '12-7.py',
 '12-8.py', '12-9.py', 'codingke', 'dist',
 'goodprogrammer', 'test.dat', 'test.py', 'test.txt']

Process finished with exit code 0
```

图 12.15 例 12-17 运行结果

从程序运行结果可以看出，该函数返回一个列表，其中的元素为 D:/1000phone 目录下所有文件名与目录名。

12.3.3 遍历目录

如果希望查看指定路径下全部子目录的所有目录和文件信息，就需要进行目录的遍历，os 模块的 walk()函数可以遍历目录树，其语法格式如下：

```
os.walk(树状结构文件夹名称)
```

该函数返回一个由 3 个元组类型的元素组成的列表，具体如下所示：

```
[(当前目录列表), (子目录列表), (文件列表)]
```

接下来演示使用 walk()函数遍历目录，如例 12-18 所示。

例 12-18 使用 walk()函数遍历目录。

```
import os # 导入 os 模块
def traversals(path):
    if not os.path.isdir(path):
        print('错误: ',path,'不是目录或不存在')
        return
    list_dirs = os.walk(path)        # os.walk 返回一个元组,包括 3 个元素
    for root, dirs, files in list_dirs: # 遍历该元组的目录和文件信息
        for d in dirs:
            print(os.path.join(root, d))  # 获取完整路径
        for f in files:
            print(os.path.join(root, f))  # 获取文件绝对路径
traversals('D:\\1000phone')
```

程序运行结束后，程序输出 D:/1000phone 目录下全部子目录的所有目录和文件信息。

12.3.4 删除目录

删除目录可以通过以下两个函数实现：

```
os.rmdir(path)          # 只能删除空目录
shutil.rmtree(path)     # 空目录、有内容的目录都可以删除
```

接下来演示这两个函数的使用，如例 12-19 所示。

例 12-19 删除目录。

```
import os, shutil  # 导入 os、shutil 模块
os.rmdir('D:/1000phone/codingke')
shutil.rmtree('D:/1000phone/goodprogrammer')
```

程序运行结束后，D:/1000phone/codingke 空目录被删除，D:/1000phone/goodprogrammer

目录及目录下内容被删除。

12.4 小 案 例

在 Windows 操作系统中，查看某个文件目录信息可以通过在右键快捷菜单中选择“属性”命令来实现，如图 12.16 所示。



图 12.16 1000phone 属性

在图 12.16 中，可以查看文件目录 1000phone 的属性。现要求编写程序，统计指定文件目录大小以及文件和文件夹数量，具体如例 12-20 所示。

例 12-20 统计指定文件目录大小以及文件和文件夹数量。

```
import os # 导入 os 模块
# 记录总大小、文件个数、目录个数
totalSize, fileNum, dirNum = 0, 0, 0
# 遍历指定目录
def traversals(path):
    global totalSize, fileNum, dirNum
```

```
    if not os.path.isdir(path):
        print('错误：', path, '不是目录或不存在')
        return
    for lists in os.listdir(path):
        sub_path = os.path.join(path, lists)
        if os.path.isfile(sub path):
            fileNum += 1  # 统计文件数量
            totalSize += os.path.getsize(sub_path)    # 文件总大小
        elif os.path.isdir(sub_path):
            dirNum += 1                      # 统计子目录数量
            traversals(sub_path)             # 递归遍历子目录
# 单位换算
def sizeConvert(size):
    K, M, G = 1024, 1024**2, 1024**3
    if size >= G:
        return str(round(size/G, 2)) + 'GB'
    elif size >= M:
        return str(round(size/M, 2)) + 'MB'
    elif size >= K:
        return str(round(size/K, 2)) + 'KB'
    else:
        return str(size) + 'Bytes'
# 输出目录位置、大小及个数
def output(path):
    if os.path.isdir(path):
        print('类型;文件夹')
    else:
        return
    print('位置:', path)
    print('大小:', sizeConvert(totalSize) +
          '('+ str(totalSize) + ' 字节)')
    print('包含:', fileNum, '个文件, ',dirNum, '个文件夹')
# 测试
if __name__=='__main__':
    path = 'D:/1000phone'
    traversals(path)
    output(path)
```

运行结果如图 12.17 所示。

从程序运行结果可看出，程序输出的信息与图 12.16 中的信息相同。

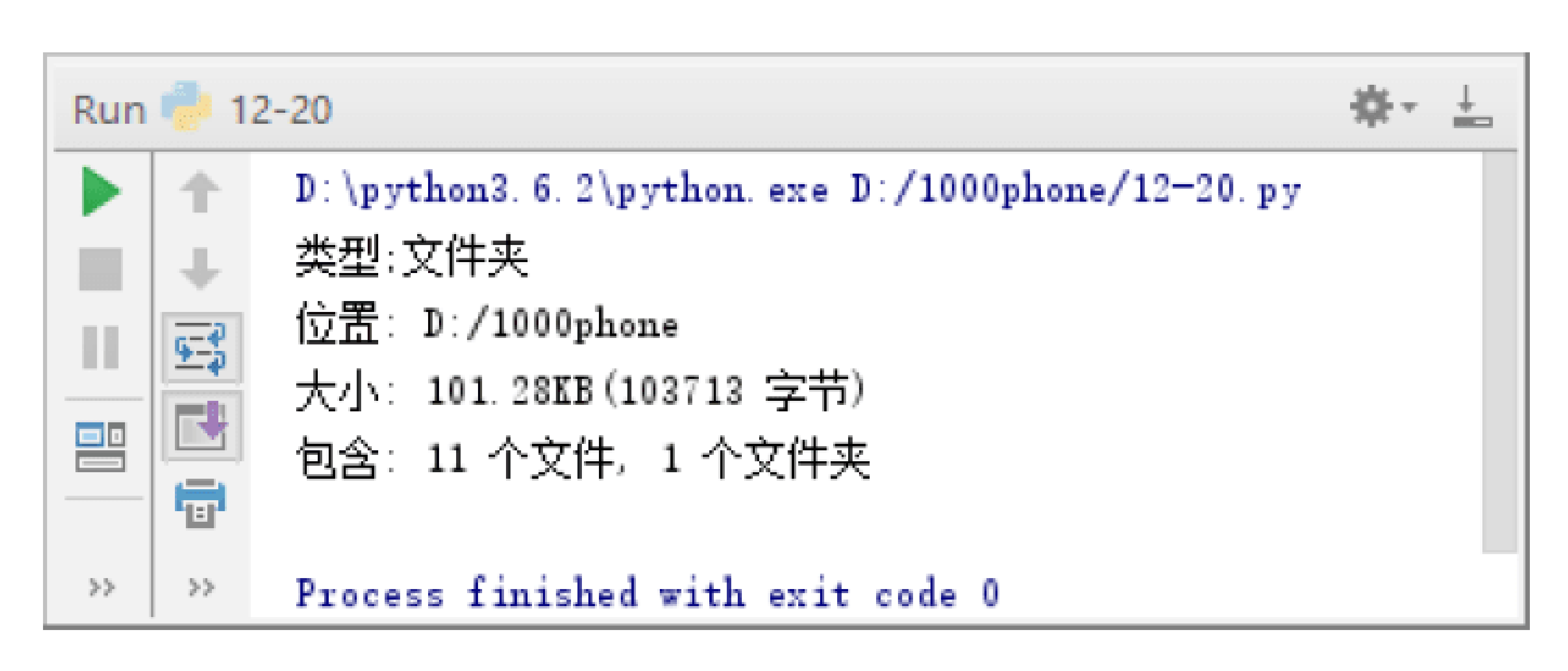


图 12.17 例 12-20 运行结果

12.5 本章小结

本章主要介绍了文件，包括文件概述、文件操作、目录操作。在实际开发中，要注意区分文本文件与二进制文件的读写操作。另外，需重点掌握目录的操作。

12.6 习 题

1．填空题

（1）文件分为文本文件与________。
（2）打开文件可以使用________函数实现。
（3）打开及关闭文件可以使用________语句实现。
（4）________方法可以获取文件指针的位置。
（5）________方法可以移动文件指针位置。

2．选择题

（1）打开一个二进制文件并进行追加写入，正确的打开模式是（ ）。

A．'ab+'　　B．'a+'
C．'w+'　　D．'wb+'

（2）下列选项中，用于读取文本文件一行内容的是（ ）。

A．文件对象.read()　　B．文件对象.readlines()
C．文件对象.readline()　　D．文件对象.read(300)

（3）os 模块的（ ）函数可以删除文件。

A．move ()　　B．del()
C．rename()　　D．remove ()

（4）os 模块的（ ）函数可以创建多级目录。

A．mkdir()　　B．make()

C．makedirs () D．makefiles()

（5）下列选项中，（ ）只能删除空目录。

A．os.rmdir(path) B．shutil.rmtree(path)

C．os.rmtree(path) D．os.remove(path)

3．思考题

（1）读取文本文件有哪些方法?

（2）简述 pickle 模块中 dump()函数与 load ()函数的作用。



4．编程题

读取文本文件 test.txt 并生成文件 newtest.txt，其中的内容与 test.txt 一致，但是在每行的首部添加了行号。

第 13 章 chapter 13

异　　常

本章学习目标

- 理解异常的概念。
- 掌握异常的处理。
- 掌握触发异常。
- 掌握自定义异常。

异常是指程序运行时引发的错误，引发错误的原因有多种，例如语法错误、除数为零、打开不存在的文件等。若这些错误没有进行处理，则会导致程序终止运行，而合理地使用异常处理错误，可以使程序具有更强的容错性。

13.1　异常概述

13.1.1　异常的概念

在生活中，使用计算机中的某个应用软件时，由于某种错误，可能会引发异常，如图 13.1 所示。



图 13.1　程序异常

在程序中，当 Python 检测到一个错误时，解释器就会指出当前流程已无法继续执行下去，这时就出现了异常。例如，使用 print()函数输出一个未定义的变量值，具体如下所示：

```
print(name)
```

在 Python 程序中，如果出现异常，而异常对象并未被捕获或处理，程序就会用自动

回溯，返回一种错误信息，并终止执行。上述语句返回的错误信息如下：

```
Traceback (most recent call last):
  File "D:/1000phone/test.py", line 1, in <module>
    print(name)
NameError: name 'name' is not defined
```

上述信息提示 name 变量名未定义，NameError 为 Python 的内建异常类。异常是指因为程序出错而在正常控制流以外采取的行为，即异常是一个事件，该事件可能会在程序执行过程中发生并影响程序的正常执行。

13.1.2 异常类

Python 为了区分不同的异常，其中内置了许多异常类，常见的异常类如表 13.1 所示。

表 13.1 常见的异常类

异常类名称	基 类	说 明
BaseException	object	所有异常类的直接或间接基类
Exception	BaseException	所有非退出异常的基类
SystemExit	BaseException	程序请求退出时抛出的异常
KeyboardInterrupt	BaseException	用户中断执行（通常是按 Ctrl+C 键）时抛出
GeneratorExit	BaseException	生成器发生异常，通知退出
ArithmeticError	Exception	所有数值计算错误的基类
FloatingPointError	ArithmeticError	浮点运算错误
OverflowError	ArithmeticError	数值运算超出最大限制
ZeroDivisionError	ArithmeticError	除零导致的异常
AssertionError	Exception	断言语句失败
AttributeError	Exception	对象没有这个属性
EOFError	Exception	读取超过文件结尾
OSError	Exception	I/O 相关错误的基类
ImportError	Exception	导入模块/对象失败
LookupError	Exception	查找错误的基类
IndexError	LookupError	序列中没有此索引
KeyError	LookupError	映射中没有这个键
MemoryError	Exception	内存溢出错误
NameError	Exception	未声明、未初始化对象
UnboundLocalError	NameError	访问未初始化的本地变量
ReferenceError	Exception	弱引用试图访问已经垃圾回收了的对象
RuntimeError	Exception	一般的运行时错误
NotImplementedError	RuntimeError	尚未实现的方法
SyntaxError	Exception	语法错误
IndentationError	SyntaxError	缩进错误
TabError	IndentationError	Tab 和空格混用

续表

异常类名称	基　类	说　明
SystemError	Exception	一般的解释器系统错误
TypeError	Exception	对类型无效的操作
ValueError	Exception	传入无效的参数
Warning	Exception	警告的基类
RuntimeWarning	Warning	可疑的运行时行为警告基类
SyntaxWarning	Warning	可疑的语法警告基类

在表 13.1 中，BaseException 是异常的顶级类，但用户定义的类不能直接继承这个类，而是要继承 Exception。Exception 类是与应用相关异常的顶层基类，除了系统退出事件类（SystemExit、KeyboardInterrupt 和 GeneratorExit）之外，几乎所有用户定义的类都应该继承自这个类，而不是 BaseException 类。

13.2　捕获与处理异常

为了防止程序运行中遇到异常而意外终止，开发时应对可能出现的异常进行捕获并处理。Python 程序使用 try、except、else、finally 这 4 个关键字来实现异常的捕获与处理。

13.2.1　try-except 语句

try-except 语句可以捕获异常并进行处理，其语法格式如下：

```
try:
    # 可能出现异常的语句
except 异常类名:
    # 处理异常的语句
```

当 try 语句块中某条语句出现异常时，程序就不再执行 try 语句块中后面的语句，而是直接执行 except 语句块，如例 13-1 所示。

例 13-1　try-except 语句。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    print(a, '/', b, '结果为', a / b)
    print('运算结束')
except ZeroDivisionError:
    print('除数不能为 0')
print('程序结束')
```

程序运行时，输入 4 与 2，则运行结果如图 13.2 所示。

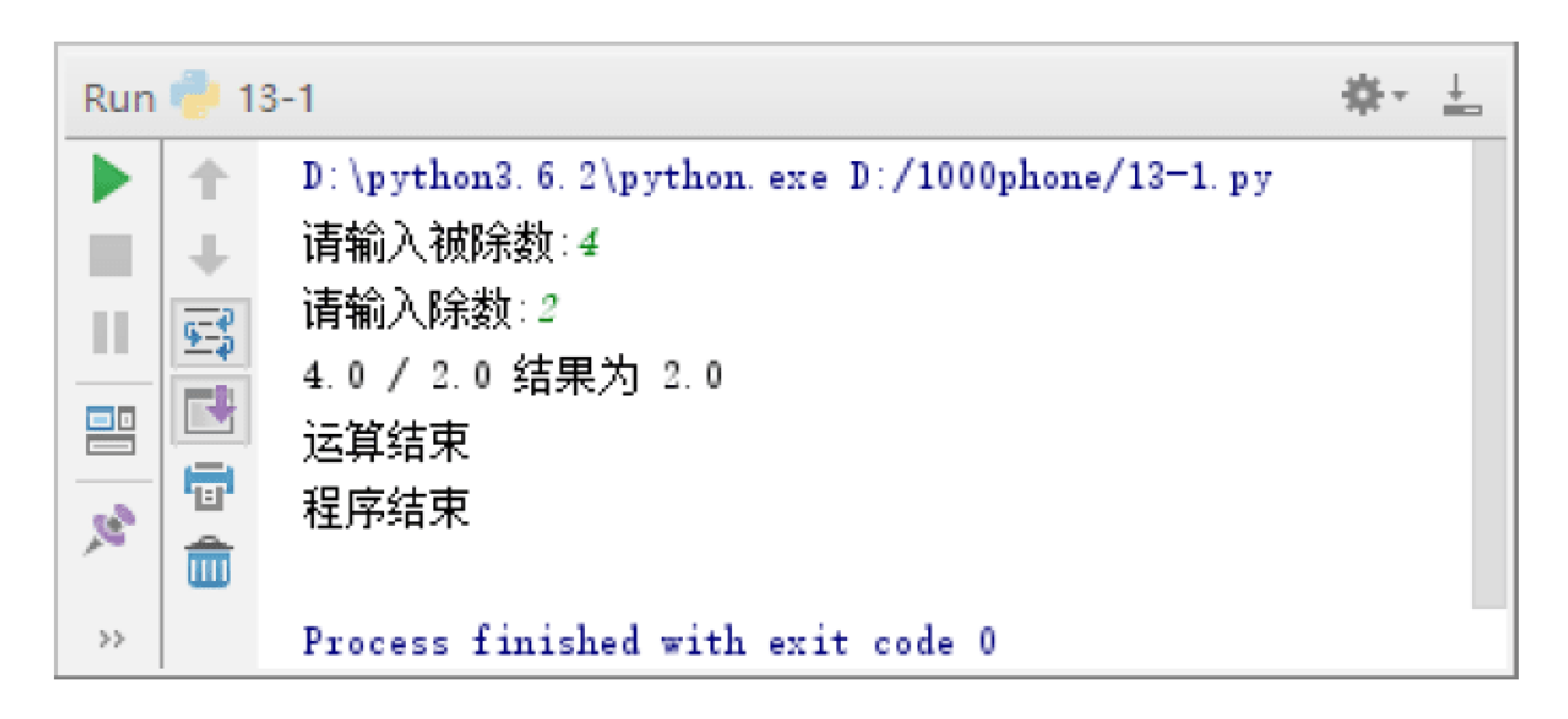


图 13.2 例 13-1 运行结果（一）

再次运行程序，输入 4 与 0，则运行结果如图 13.3 所示。

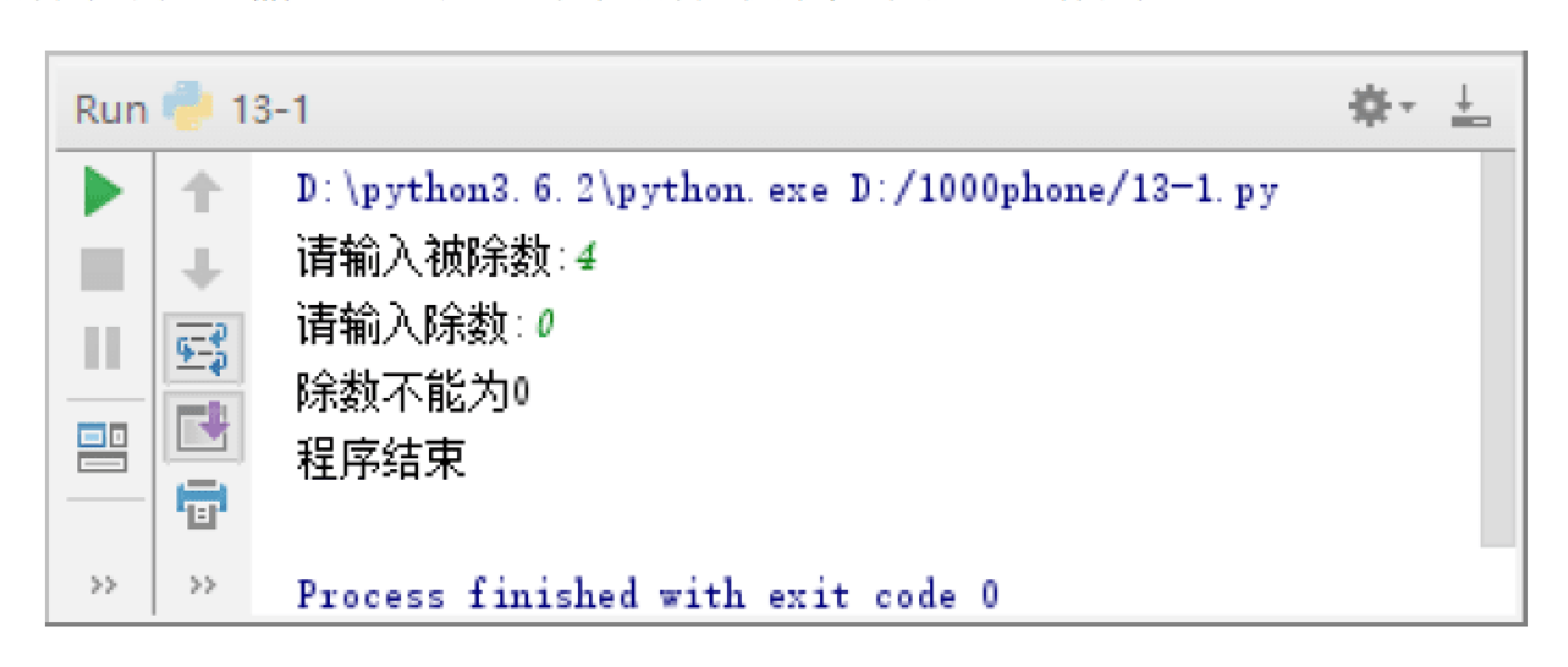


图 13.3 例 13-1 运行结果（二）

从两次运行结果可看出，程序没有触发异常与触发异常执行的流程并不一致。程序中一旦发生异常，就不会执行 try 语句块中剩余的语句，而是直接执行 except 语句块。另外，本程序捕获并处理了异常，因此，当输入的除数为 0 时，程序可以正常结束，而不是终止运行。

需要注意的是，例 13-1 程序只能捕捉 except 后面的异常类，如果发生其他类型的异常，程序依然会终止。例如，运行例 13-1 的程序，输入 ab 再回车，则程序出现错误，如图 13.4 所示。

```
Run 13-1
D:\python3.6.2\python.exe D:/1000phone/13-1.py
请输入被除数:ab
Traceback (most recent call last):
  File "D:/1000phone/13-1.py", line 2, in <module>
    a = float(input('请输入被除数:'))
ValueError: could not convert string to float: 'ab'

Process finished with exit code 1
```

图 13.4 错误信息

在图 13.4 中，错误信息提示字符串类型不能转化为浮点型。为了保证程序正常运行，此时就需要捕获并处理多种异常，其语法格式如下：

```
try:
    # 可能出现异常的语句
except 异常类名 1:
    # 处理异常 1 的语句
except 异常类名 2:
    # 处理异常 2 的语句
...
```

接下来演示捕获并处理多种异常，如例 13-2 所示。

例 13-2 捕获并处理多种异常。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    print(a, '/', b, '结果为', a / b)
    print('运算结束')
except ZeroDivisionError:
    print('除数不能为 0')
except ValueError:
    print('传入参数无效')
print('程序结束')
```

程序运行时，输入 ab 并回车，则运行结果如图 13.5 所示。

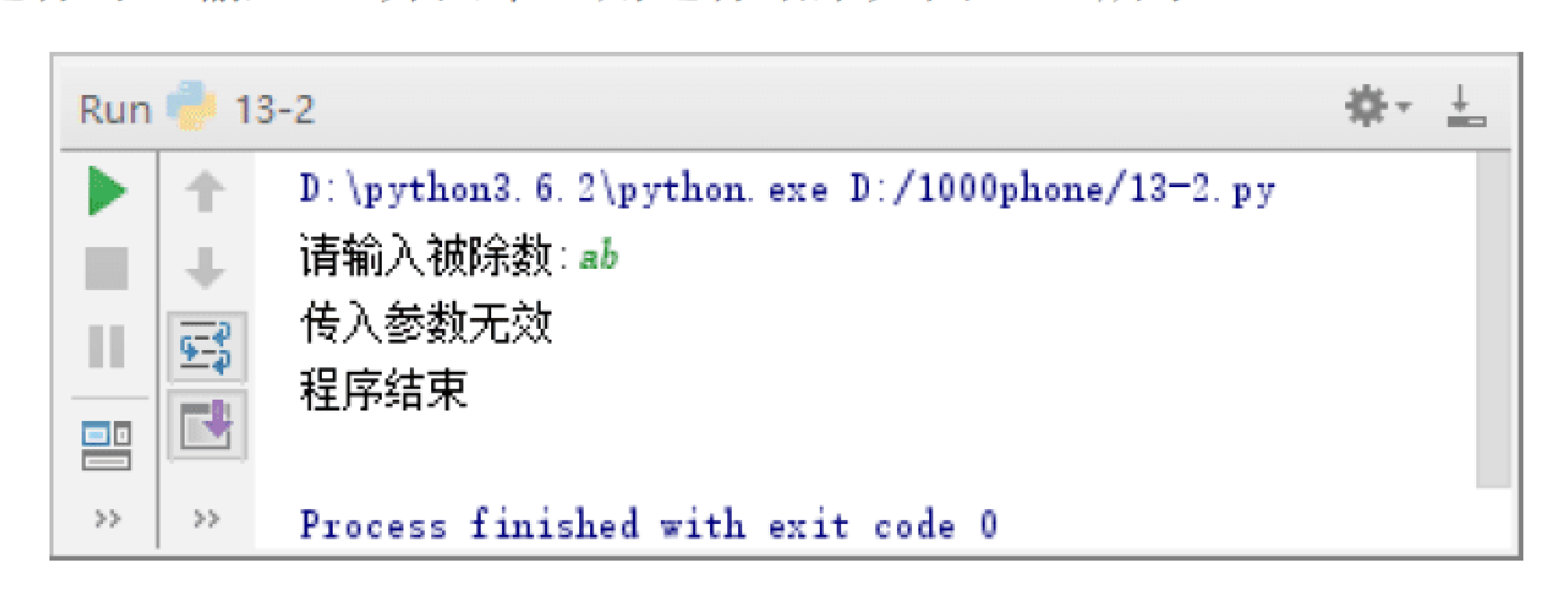


图 13.5 例 13-2 运行结果

在例 13-2 中，程序中增加了对 ValueError 异常的处理。在程序中，虽然开发者可以编写处理多种异常的代码，但异常是防不胜防的，很有可能再出现其他异常，此时就需要捕获并处理所有可能发生的异常，其语法格式如下：

```
try:
    # 可能出现异常的语句
except 异常类名:
    # 处理异常的语句
except:
```

```
        # 与上述异常不匹配时，执行此语句块
```

如果程序发生了异常，但是没有找到匹配的异常类别，则执行不带任何匹配类型的 except 语句块。

接下来演示捕获并处理所有异常，如例 13-3 所示。

例 13-3 捕获并处理所有异常。

```
1   try:
2       a = float(input('请输入被除数:'))
3       b = float(input('请输入除数:'))
4       print(a, '/', b, '结果为', a / b)
5       print('运算结束')
6   except ZeroDivisionError:
7       print('除数不能为 0')
8   except:
9       print('其他错误')
10  print('程序结束')
```

程序运行时，输入 ab 并回车，则运行结果如图 13.6 所示。

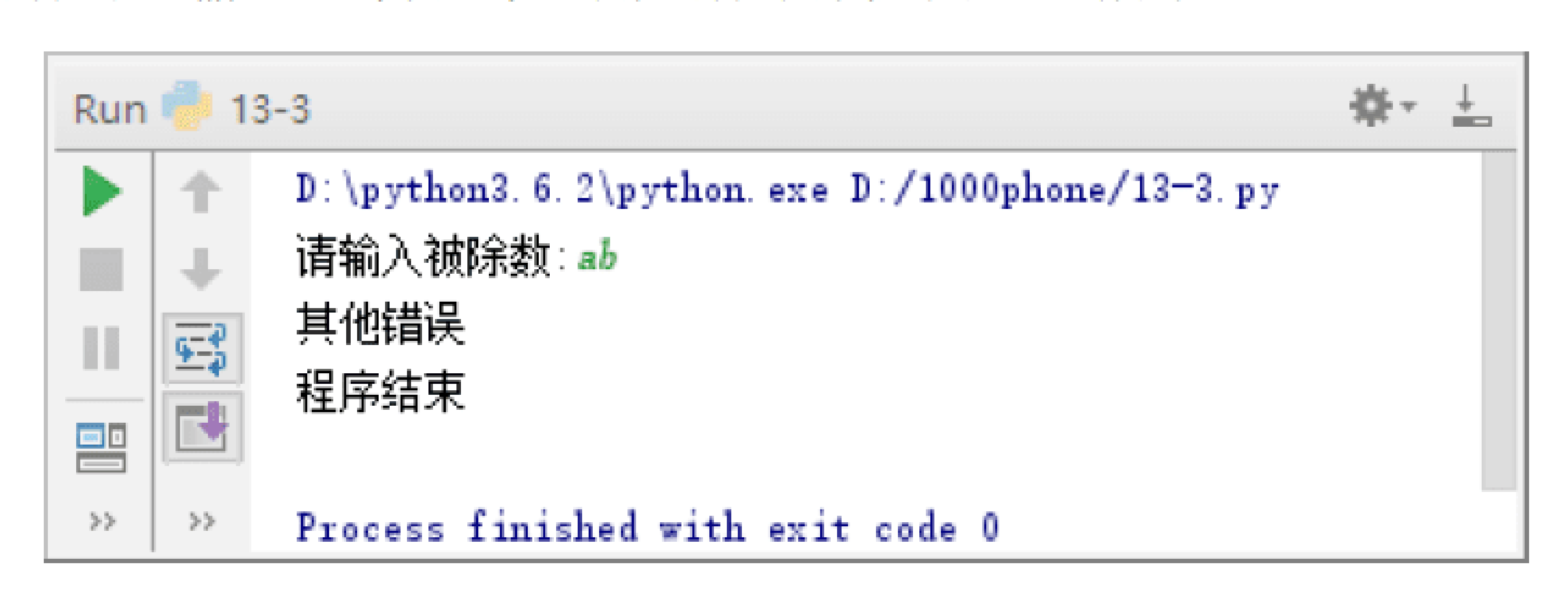


图 13.6 例 13-3 运行结果

在例 13-3 中，第 8 行通过 except 语句处理除 ZeroDivisionError 异常外的其他所有异常。

13.2.2 使用 as 获取异常信息

为了区分不同的异常，可以使用 as 关键字来获取异常信息，其语法格式如下：

```
try:
    # 可能出现异常的语句
except 异常类名 as 异常对象名:
    # 处理异常的语句
```

通过异常对象名便可以访问异常信息，如例 13-4 所示。

例 13-4 通过异常对象名访问异常信息。

```
try:
    a = float(input('请输入被除数:'))
```

```
3       b = float(input('请输入除数:'))
4       print(a, '/', b, '结果为', a / b)
5       print('运算结束')
6   except ZeroDivisionError as e:
7       print(type(e), e)
8   print('程序结束')
```

运行结果如图 13.7 所示。

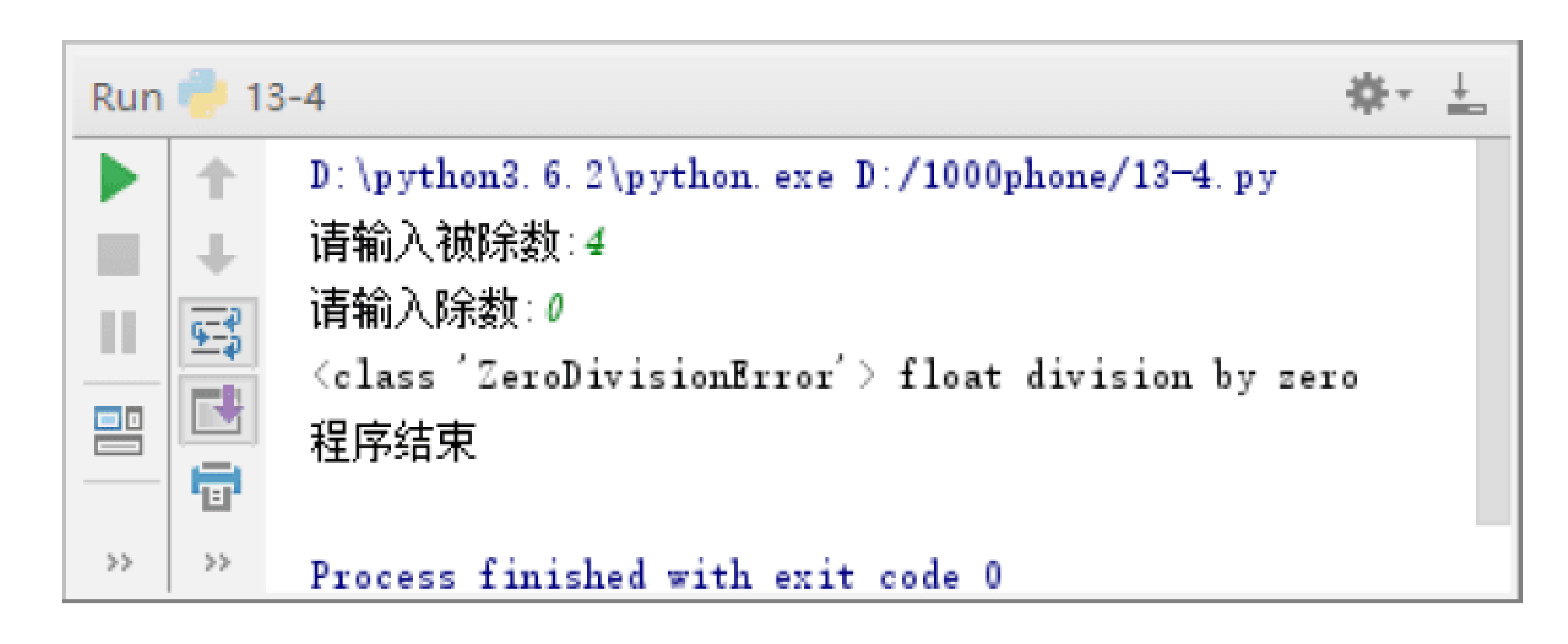


图 13.7　例 13-4 运行结果

在例 13-4 中，第 6 行通过 as 关键字可以获取 ZeroDivisionError 异常类的实例对象 e，第 7 行通过 print()函数打印异常信息。

如果程序需要获取多种异常信息，则可以使用如下语法格式：

```
try:
    # 可能出现异常的语句
except (异常类名1, 异常类名2, …) as 异常对象名:
    # 处理异常的语句
```

接下来演示获取多种异常信息，如例 13-5 所示。

例 13-5　获取多种异常信息。

```
1   try:
2       a = float(input('请输入被除数:'))
3       b = float(input('请输入除数:'))
4       print(a, '/', b, '结果为', a / b)
5       print('运算结束')
6   except (ZeroDivisionError, ValueError) as e:
7       print(type(e), e)
8   print('程序结束')
```

程序运行时，输入 4 与 0，则运行结果如图 13.8 所示。

再次运行程序，输入 ab，则运行结果如图 13.9 所示。

从上述运行结果可看出，当程序出现 ZeroDivisionError 或 ValueError 异常时，第 6 行语句会自动捕获相应的异常并生成该异常类的实例对象。

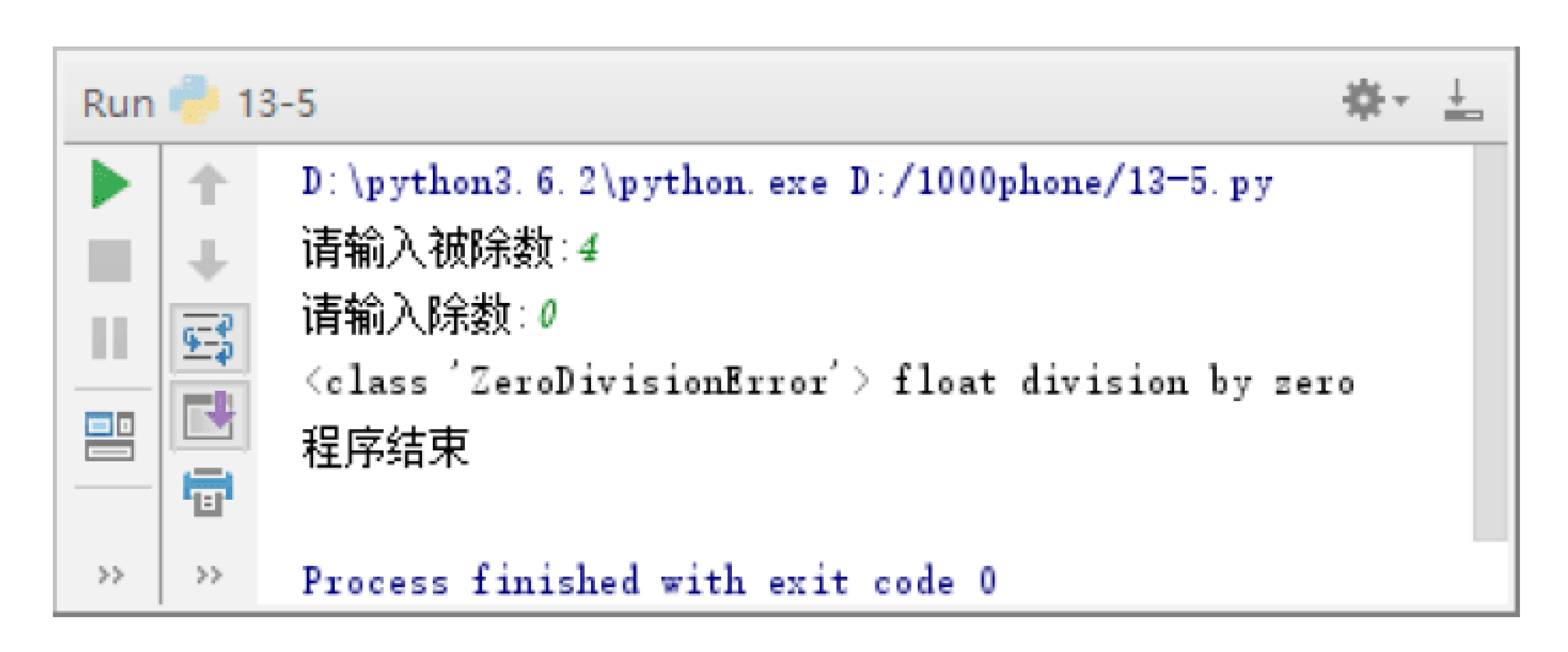


图 13.8 例 13-5 运行结果（一）

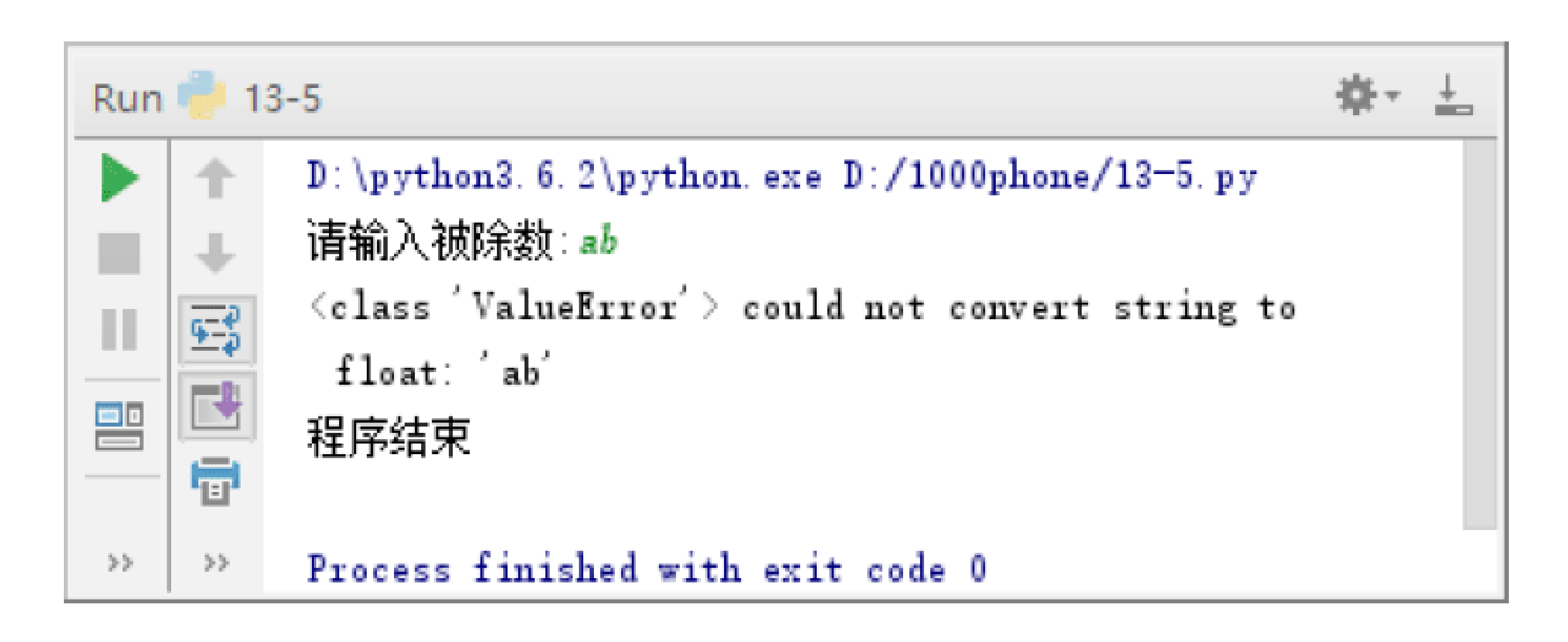


图 13.9 例 13-5 运行结果（二）

如果程序需要获取所有异常信息，则可以使用如下语法格式：

```
try:
    # 可能出现异常的语句
except BaseException as 异常对象名:
    # 处理异常的语句
```

所有的异常类都继承自 BaseException 类，因此上述语句可以获取所有异常信息，如例 13-6 所示。

例 13-6 获取所有异常信息。

```
1  try:
2      a = float(input('请输入被除数:'))
3      b = float(input('请输入除数:'))
4      print(a, '/', b, '结果为', a / b)
5      print('运算结束')
6  except BaseException as e:
7      print(type(e), e)
8  print('程序结束')
```

运行结果如图 13.10 所示。

在例 13-6 中，第 6 行语句可以获取所有异常，但不建议在程序中直接捕获所有异常，因为它会隐藏所有程序员未想到并且未做好准备处理的错误。

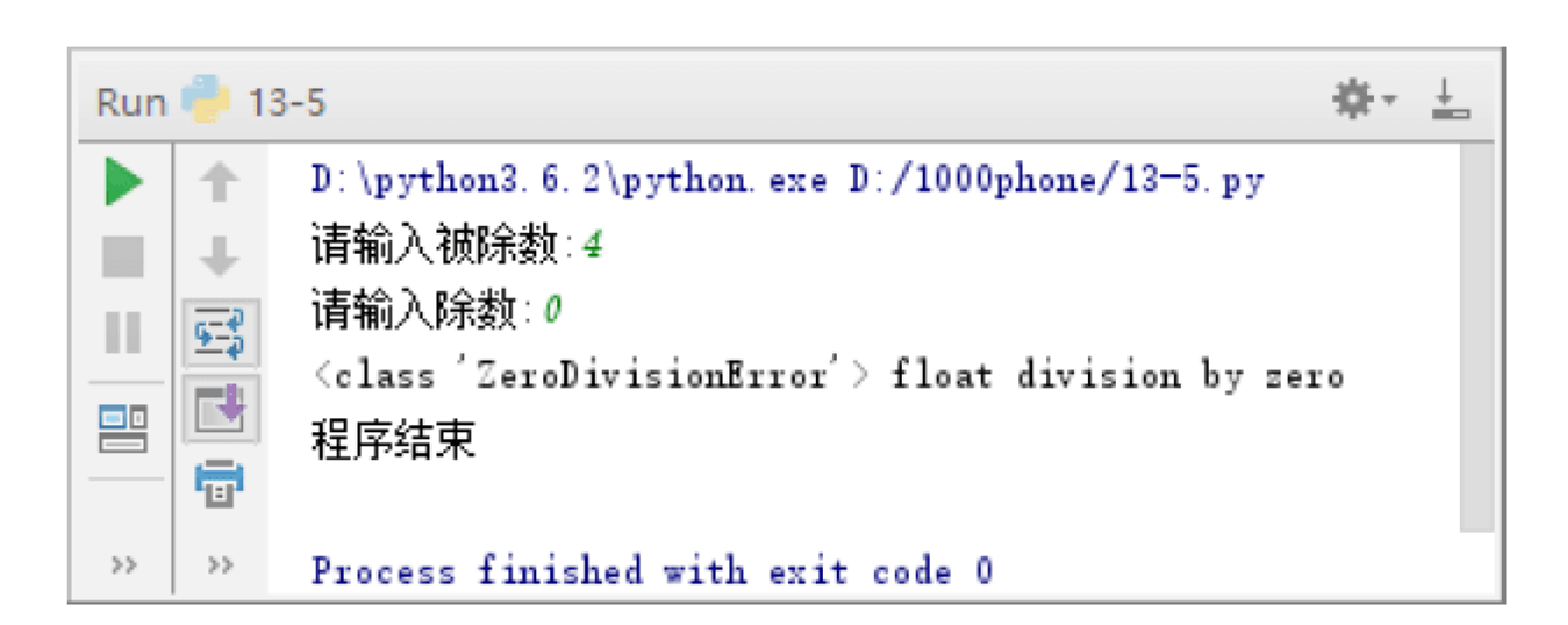


图 13.10 例 13-6 运行结果

13.2.3 try-except-else 语句

try-except-else 语句用于处理未捕获到异常的情形，其语法格式如下：

```
try:
    # 可能出现异常的语句
except BaseException as 异常对象名:
    # 处理异常的语句
else:
    # 未捕获到异常执行的语句
```

如果 try 语句内出现了异常，则执行 except 语句块，否则执行 else 语句块。

接下来演示 try-except-else 语句的用法，如例 13-7 所示。

例 13-7 try-except-else 语句的用法。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    result = a / b
except BaseException as e:
    print(type(e), e)
else:
    print(a, '/', b, '结果为', result)
print('程序结束')
```

程序运行时，输入 4 与 0，则运行结果如图 13.11 所示。

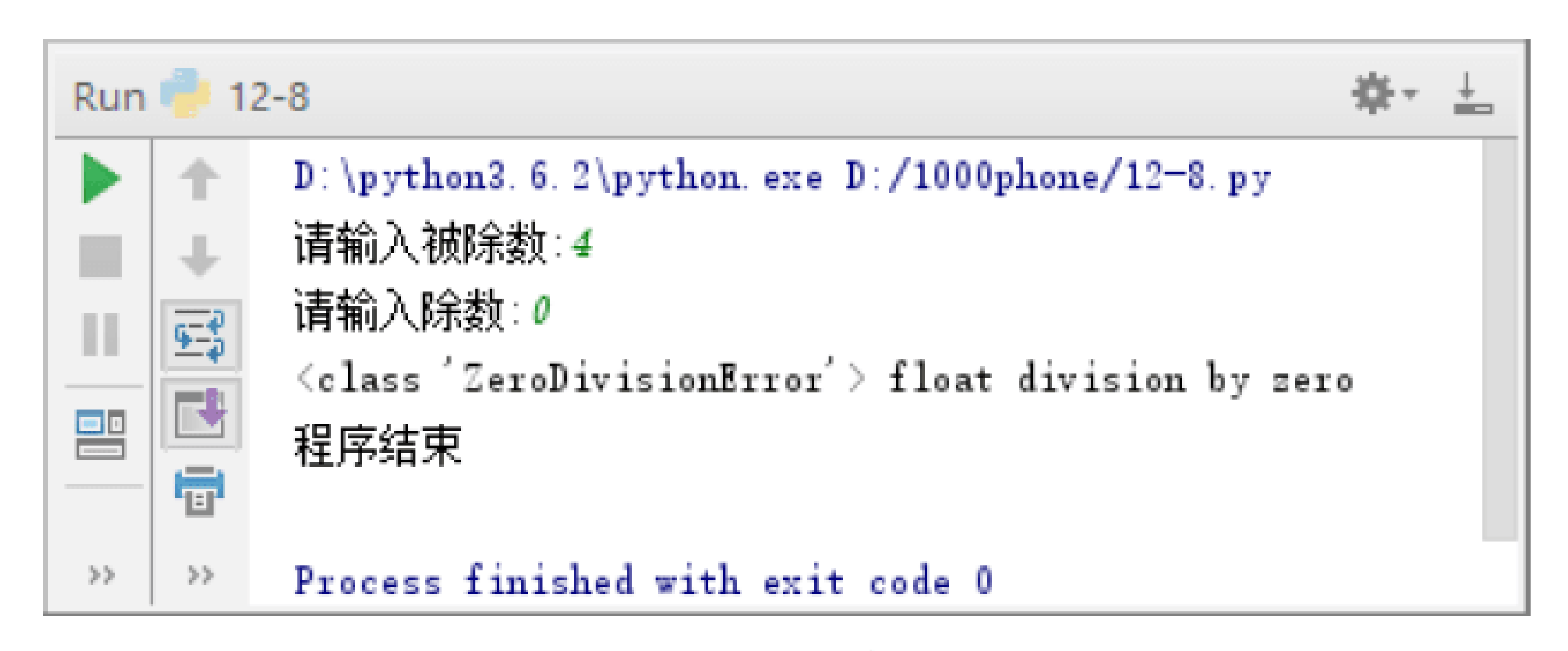


图 13.11 例 13-7 运行结果（一）

再次运行程序，输入 4 与 2，则运行结果如图 13.12 所示。

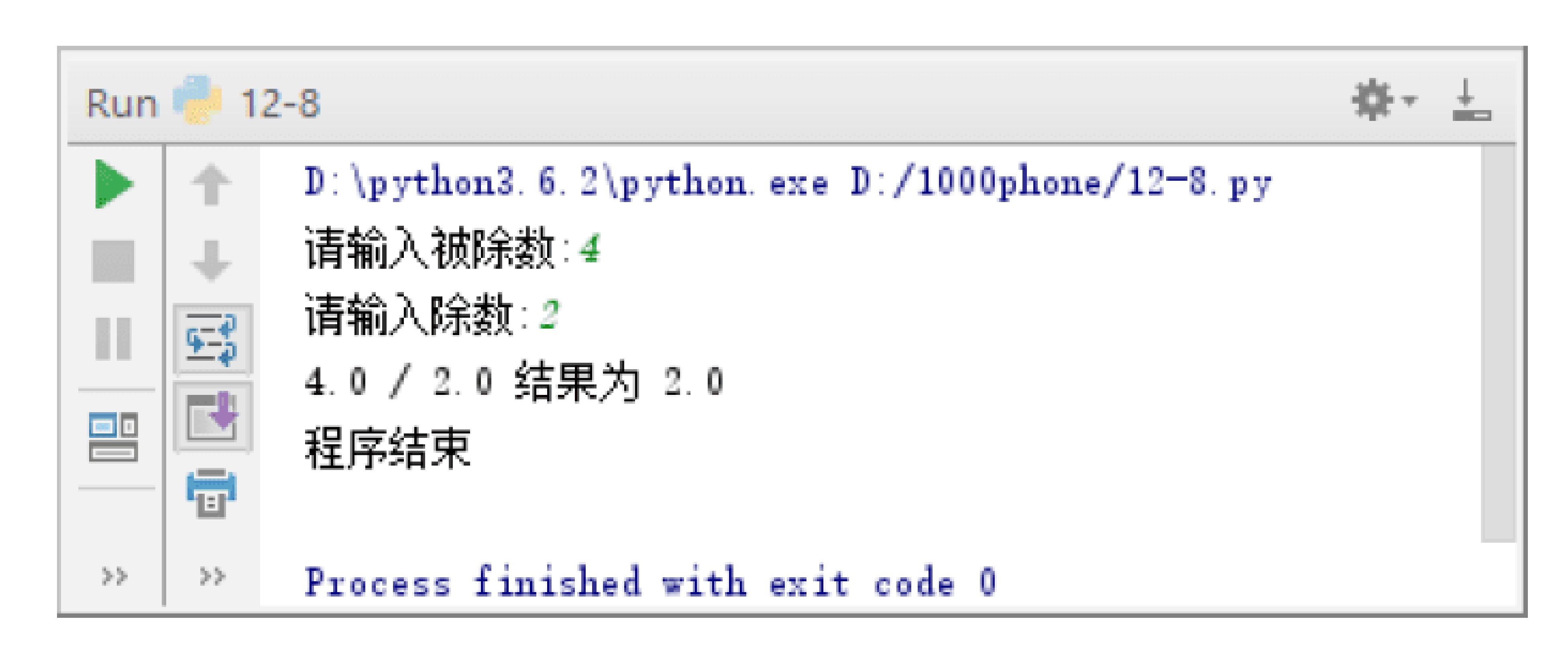


图 13.12　例 13-7 运行结果（二）

在例 13-7 中，当未捕获到异常时，程序执行 else 语句块。

13.2.4　try-finally 语句

在 try-finally 语句中，无论 try 语句块中是否发生异常，finally 语句块中的代码都会执行，其语法格式如下：

```
try:
    # 可能出现异常的语句
finally:
    # 无论是否发生异常都会执行的语句
```

其中，finally 语句块用于清理在 try 块中执行的操作，如释放其占有的资源（如文件对象、数据库连接、图形句柄等）。

接下来演示 try-finally 语句的用法，如例 13-8 所示。

例 13-8　try-finally 语句的用法。

```
try:
    f = open('test.txt', 'a+')
    i = 1
    while True:
        str = input('请输入第%d 行字符串（按 Q 结束）：'%i)
        if str.upper() == 'Q':
            break
        f.write(str + '\n')
        i += 1
except KeyboardInterrupt:
    print('程序中断！(Ctrl+C) ')
finally:
    f.close()
```

```
    print('文件关闭')
print('程序结束')
```

打开控制台（按 Window+R 组合键打开运行窗口，在输入框中输入 cmd 并单击“确定”按钮），在命令行模式下进入 D:\1000phone 目录，输入“python 13-8.py”，开始执行程序，如图 13.13 所示。

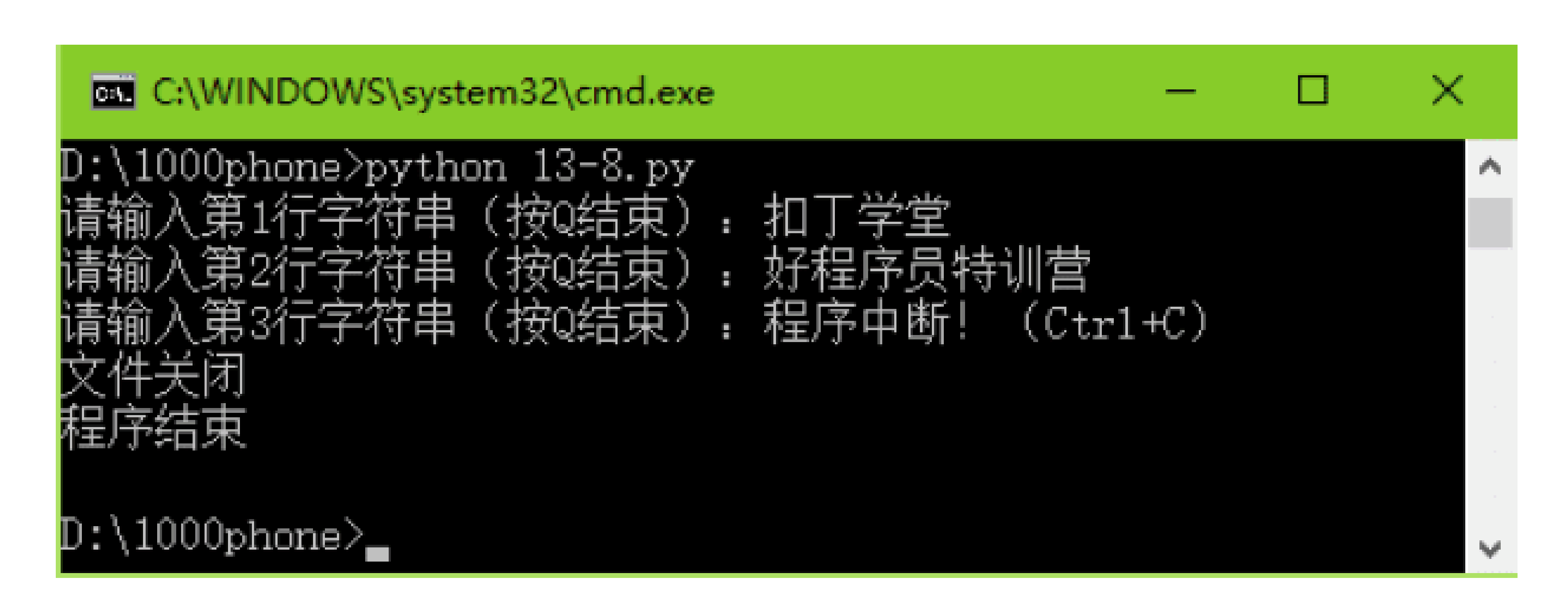


图 13.13　在命令行模式中执行程序

当提示输入第 3 行字符串时，在键盘中按下 Ctrl+C 键，此时引发 KeyboardInterrupt 异常，程序立即执行 except 语句，之后再执行 finally 语句。

另外，with-as 语句可作为 try-finally 语句处理异常的替代，其语法格式如下：

```
with 表达式 [as 变量名]:
    with 语句块
```

该语句用于定义一个有终止或清理行为的情况，如释放线程资源、文件、数据库连接等，在这些场合下使用 with 语句将使代码更加简洁。

在讲解文件打开与关闭时，本书使用的就是 with-as 语句。with 后面的表达式的结果将生成一个支持环境管理协议的对象，该对象中定义了__enter__() 和__exit__()方法。在 with 内部的语句块执行之前，调用__enter__()方法运行构造代码，如果在 as 后面指定了一个变量，则将返回值和这个变量名绑定。当 with 内部语句块执行结束后，自动调用__exit__()方法，同时执行必要的清理工作，不管执行过程中有无异常发生。

以上学习了 try-except 语句、try-except-else 语句和 try-finally 语句，在实际开发中，经常需要将 3 种语句结合起来使用，具体如下所示：

```
try:
    # 可能出现异常的语句
except 异常类名 as 异常对象名:
    # 处理特定异常的语句
except:
    # 处理多个异常的语句
else:
    # 未捕获到异常执行的语句
finally:
    # 无论是否发生异常都会执行的语句
```

程序先执行 try 语句块，若 try 语句块中的某一语句执行时发生异常，则程序跳转到 except 语句，从上到下判断抛出的异常是否与 except 后面的异常类相匹配，并执行第一个匹配该异常的 except 后面的语句块。

若 try 语句块中发生了异常，但是没有找到匹配的异常类，则执行不带任何匹配类型的 except 语句块。

若没有发生任何异常，则程序在执行完 try 语句块后直接进入 else 语句块。

最后，无论程序是否发生异常，都会执行 finally 语句块。

13.3 触发异常

触发异常有两种情况：一种是程序执行中因为错误自动触发异常，另一种是显式地使用 raise 或 assert 语句手动触发异常。Python 捕获与处理这两种异常的方式是相同的。本节主要介绍手动触发异常。

13.3.1 raise 语句

raise 语句可以手动触发异常，其使用方法有如下 3 种。

1. 通过类名触发异常

该方法只需指明异常类便可创建异常类的实例对象并触发异常，其语法格式如下：

```
raise 异常类名
```

例如，手动触发语法错误异常，则可以使用以下语句：

```
raise SyntaxError
```

程序运行时，输出以下信息：

```
Traceback (most recent call last):
  File "D:/1000phone/test.py", line 1, in <module>
    raise SyntaxError
SyntaxError: None
```

2. 通过异常类的实例对象触发异常

该方法只需指明异常类的实例对象便可触发异常，其语法格式如下：

```
raise 异常类的实例对象
```

例如，手动触发除零导致的异常，则可以使用以下语句：

```
raise ZeroDivisionError()
```

程序运行时，输出以下信息：

```
Traceback (most recent call last):
  File "D:/1000phone/test.py", line 1, in <module>
    raise ZeroDivisionError()
ZeroDivisionError
```

此外，该方法还可以指定异常信息，具体如下所示：

```
raise ZeroDivisionError('除数为零!')
```

程序运行时，输出以下信息：

```
Traceback (most recent call last):
  File "D:/1000phone/test.py", line 1, in <module>
    raise ZeroDivisionError('除数为零!')
ZeroDivisionError: 除数为零!
```

3．抛出异常

raise 语句还可以抛出异常，具体如下所示：

```
try:
    raise ZeroDivisionError
except:
    print('捕捉到异常!')
    raise         # 重新触发刚才发生的异常
```

程序运行时，输出以下信息：

```
Traceback (most recent call last):
捕捉到异常!
  File "D:/1000phone/test.py", line 2, in <module>
    raise ZeroDivisionError
```

可以看出，程序执行了 except 语句块中的代码，其中的 raise 语句会重新触发 ZeroDivisionError 异常，但此时异常对象并未被捕获或处理，因此程序终止运行。

13.3.2 assert 语句

assert 语句（又称断言）是有条件的触发异常，其语法格式如下：

```
assert 表达式 [, 参数]
```

其中，当表达式为真时，不触发异常；当表达式为假时，触发 AssertionError 异常。若给定了参数部分，则在 AssertionError 后将参数部分作为异常信息的一部分给出。

assert 语句的主要功能是帮助程序员调试程序，以保证程序运行的正确性，因此它

一般在开发调试阶段使用。

接下来演示 assert 语句的用法，如例 13-9 所示。

例 13-9 assert 语句的用法。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    assert a >= b, '被除数大于除数'
    result = a / b
except BaseException as e:
    print(e.__class__.__name__, ':', e)
print('程序结束')
```

程序运行时，输入 4 与 2，则运行结果如图 13.14 所示。

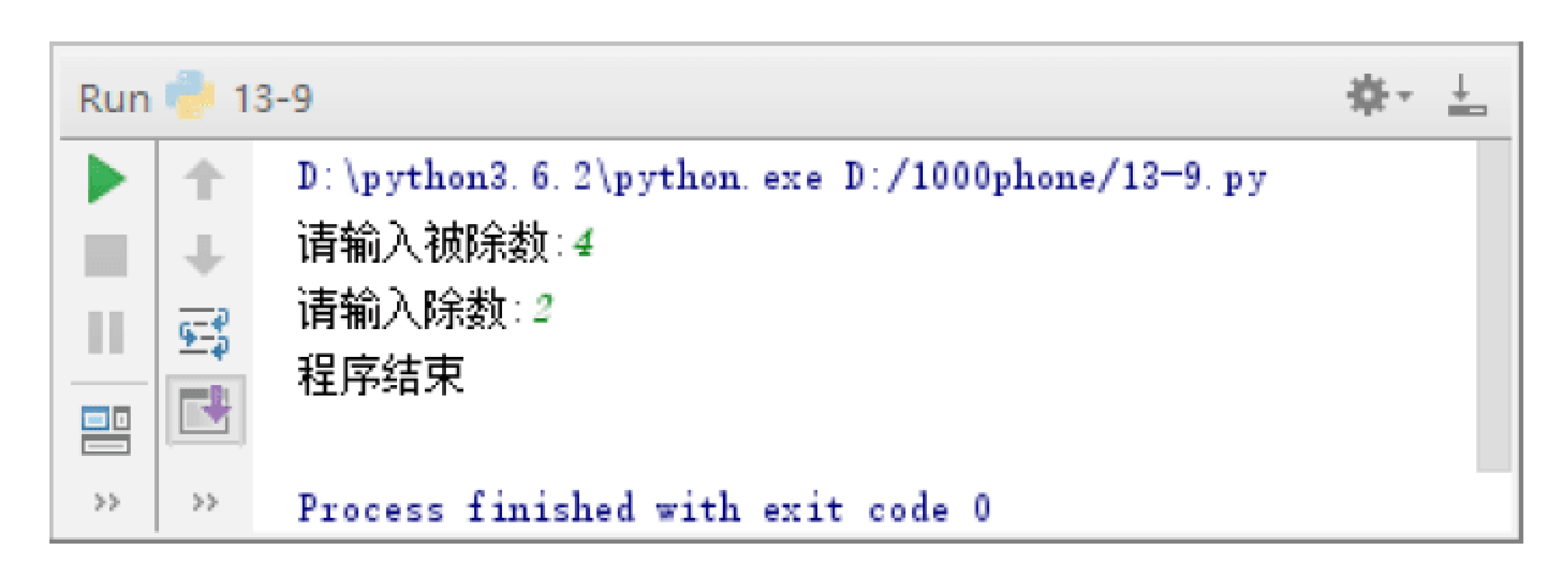


图 13.14 例 13-9 运行结果（一）

再次运行程序，输入 2 与 4，则运行结果如图 13.15 所示。

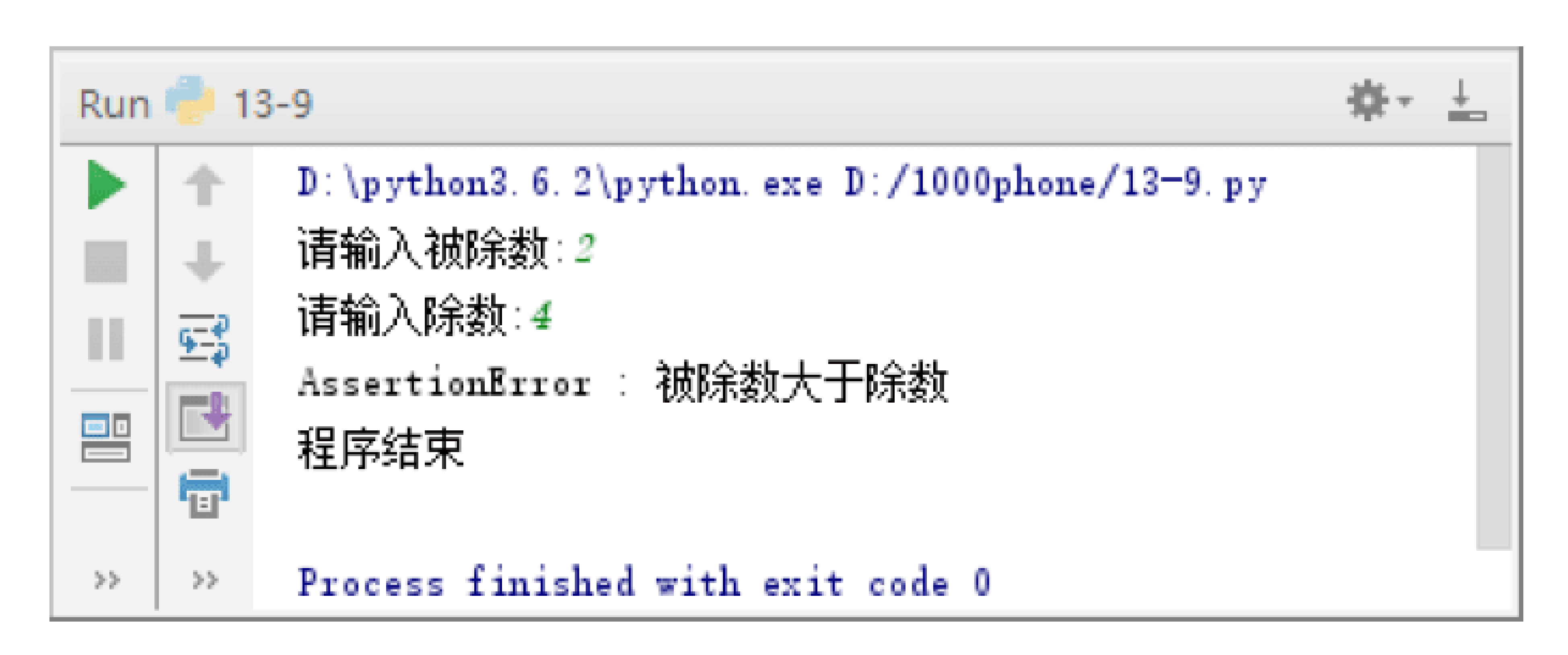


图 13.15 例 13-9 运行结果（二）

在例 13-9 中，只有当输入的被除数小于除数时，程序才会触发 AssertionError 异常。

13.4 自定义异常

Python 中内置的异常类毕竟有限，用户有时须根据需求设置其他异常，如学生成绩

不能为负数、限定密码长度等。自定义异常类一般继承于 Exception 或其子类，其命名一般以 Error 或 Exception 为后缀，如例 13-10 所示。

例 13-10 自定义异常。

```
1  class NumberError(Exception):    # 自定义异常类，继承于 Exception
2      def __init__(self, data = ''):
3          Exception.__init__(self, data)
4          self.data = data
5      def __str__(self):            # 重载__str__方法
6          return self.__class__.__name__ + ':' + self.data + '非法数值(<= 0)'
7  try:
8          num = input('请输入正数:')
9          if float(num) <= 0:
10             raise NumberError(num) # 触发异常
11         print('输入的正数为:', num)
12 except BaseException as e:
13     print(e)
```

运行结果如图 13.16 所示。

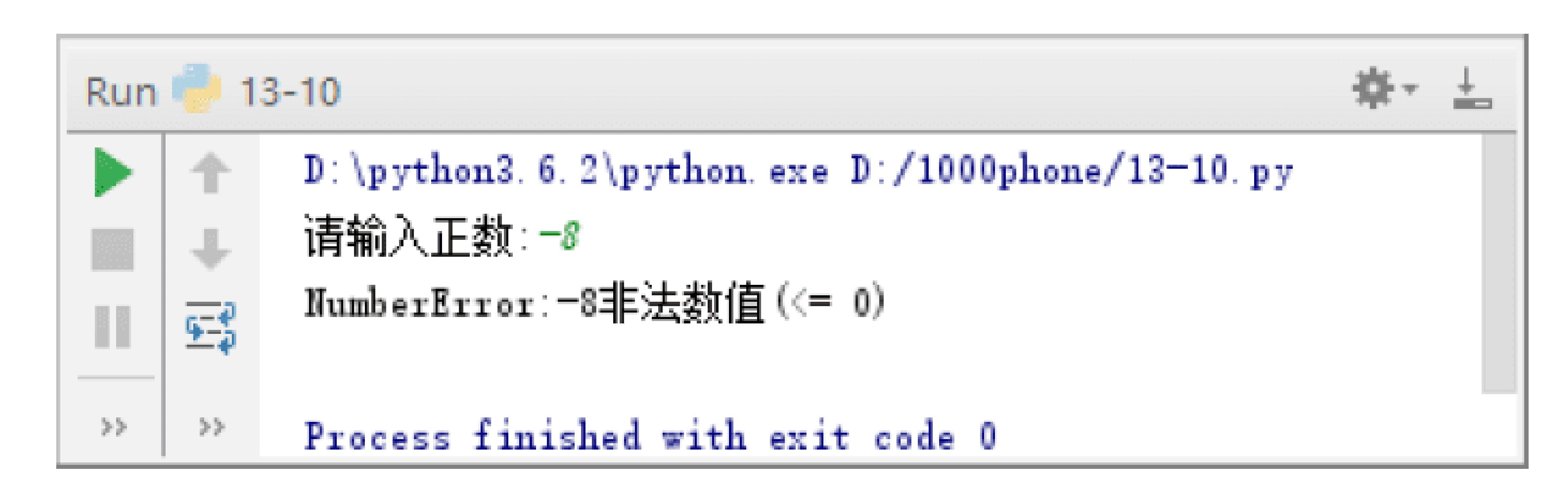


图 13.16 例 13-10 运行结果

在例 13-10 中，第 1 行自定义异常类 NumberError，继承自 Exception，第 10 行通过 raise 语句手动触发 NumberError 异常。

13.5 回溯最后的异常

当触发异常时，Python 可以回溯异常并提示许多信息，这可能会给程序员定位异常位置带来不便，因此，Python 中可以使用 sys 模块中的 exc_info()函数来回溯最后一次异常信息，该函数返回一个元组(type, value/message, traceback)，每个元素的具体含义如下所示：

- type：异常的类型；
- value/message——异常的信息或者参数；
- traceback——包含调用栈信息的对象。

接下来演示该函数的用法，如例 13-11 所示。

例 13-11 sys 模块中 exc_info()函数的用法。

```
1   import sys # 导入 sys 模块
2   try:
3       4 / 0
4   except:
5       tuple  = sys.exc_info()
6       print(tuple)
```

运行结果如图 13.17 所示。

```
Run 13-11
D:\python3.6.2\python.exe D:/1000phone/13-11.py
(<class 'ZeroDivisionError'>, ZeroDivisionError
 ('division by zero',), <traceback object at
 0x02A70E68>)

Process finished with exit code 0
```

图 13.17 例 13-11 运行结果

在例 13-11 中，第 6 行输出 sys.exc_info()函数的返回值。该函数虽然可以获取最后触发异常的信息，但是难以直接确定触发异常的代码位置。

13.6 小 案 例

计算 test.txt 文件中每行数字的总和与平均值，该文件可能不存在或为空，也可能某行不包含数字。程序中须捕获并处理可能发生的异常（不能使用 with-as 语句），具体实现如例 13-12 所示。

例 13-12 计算 test.txt 文件中每行数字的总和与平均值。

```
sum, num, flag = 0, 0, True
try:
    file = open('test.txt', 'r')
except FileNotFoundError as e:  # 处理文件不存在的异常
    print(e.__class__.__name__, ':', e)
    flag = False
if flag:
    try:
        for line in file:
            num += 1
            sum += float(line)
```

```
        ave = sum / num
    except ValueError as e:  # 处理某行不是数字的异常
        print(num, '行 ', e.__class__.__name__, ':', e)
        if num > 1:
            print(num, '行之前:')
            print('总和:', sum, ' 平均值:', sum / (num - 1))
        else:
            print('无法计算')
    except ZeroDivisionError:  # 处理空文件的异常
        print('空文件')
    else:
        print('总和:', sum, ' 平均值:', ave)
    finally:
        file.close()
print('程序结束')
```

若 test.txt 文件不存在，则程序运行结果如图 13.18 所示。

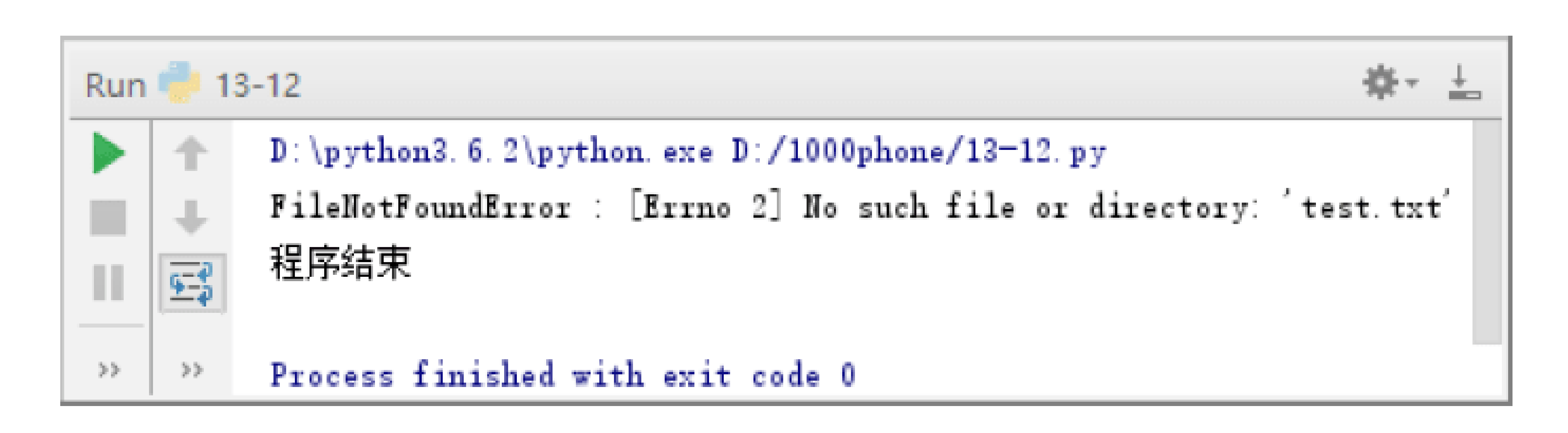


图 13.18　例 13-12 运行结果（一）

若 test.txt 文件内容如下所示：

```
12
76
扣丁学堂
23
```

则程序运行结果如图 13.19 所示。

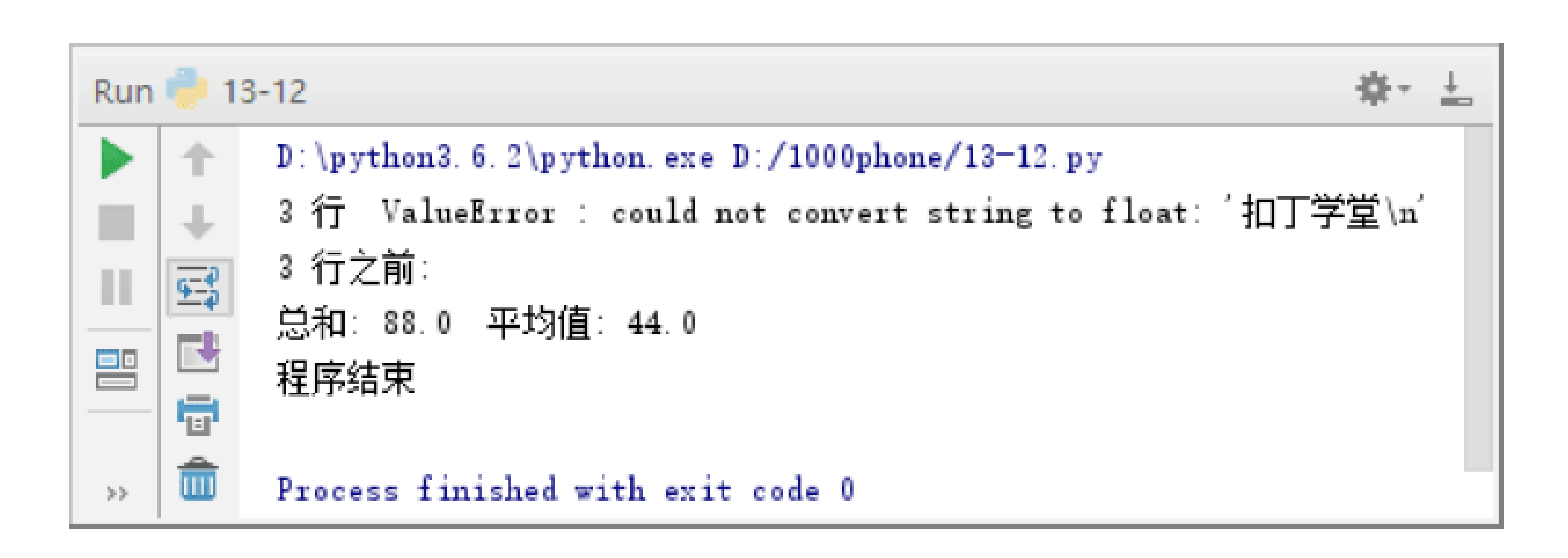


图 13.19　例 13-12 运行结果（二）

在例 13-12 中，程序通过 try-except 语句和 try-except-else-finally 语句分别对文件操作和除法操作中可能出现的异常进行捕获和处理。

13.7 本章小结

本章主要介绍了异常，包括异常的概念、触发异常、捕获与处理异常、自定义异常、回溯最后的异常。学习完本章知识，须理解异常处理的作用，即它使程序能够正常执行，不至于因异常导致退出或崩溃。

13.8 习题

1．填空题

（1）用户自定义的异常类须继承________。

（2）Exception 类的基类是________。

（3）________语句是有条件的触发异常。

（4）sys 模块中________函数用来回溯最后一次异常信息。

（5）通过________关键字可以获取异常信息。

2．选择题

（1）下列选项中，（　　）语句可以手动触发异常。

A．try　　B．except
C．raise　　D．finally

（2）当 try 语句块中未触发异常时，（　　）语句块不会执行。

A．finally　　B．except
C．else　　D．try

（3）下列选项中，语句顺序正确的是（　　）。

A．try-else-except　　B．try-except-finally-else
C．try-finally-except　　D．try-except-else-finally

（4）无论 try 语句块中是否发生异常，（　　）语句块都会被执行。

A．except　　B．except-as
C．finally　　D．else

（5）SyntaxError 表示出现（　　）异常。

A．语法错误　　B．缩进错误
C．除零错误　　D．断言

3．思考题

（1）简述异常处理语句的执行流程。

（2）简述 raise 语句的使用方法。

4．编程题



输入与输出员工的姓名、年龄、月收入（输出年收入），假设姓名字符串长度为 2～18，年龄为 18～60，月收入大于 2500，如果不满足上述条件，则手动触发异常并处理。

第 14 章 chapter 14

综 合 案 例

本章学习目标

- 了解程序的开发流程。
- 掌握程序的流程控制。

通过前面的学习，相信大家已掌握 Python 语言基础知识。为了提高大家的动手能力，本章将回顾一下所学知识，设计一款 2048 游戏。

14.1 需 求 分 析

2048 游戏是一款数字益智游戏，如图 14.1 所示。具体游戏规则如下：

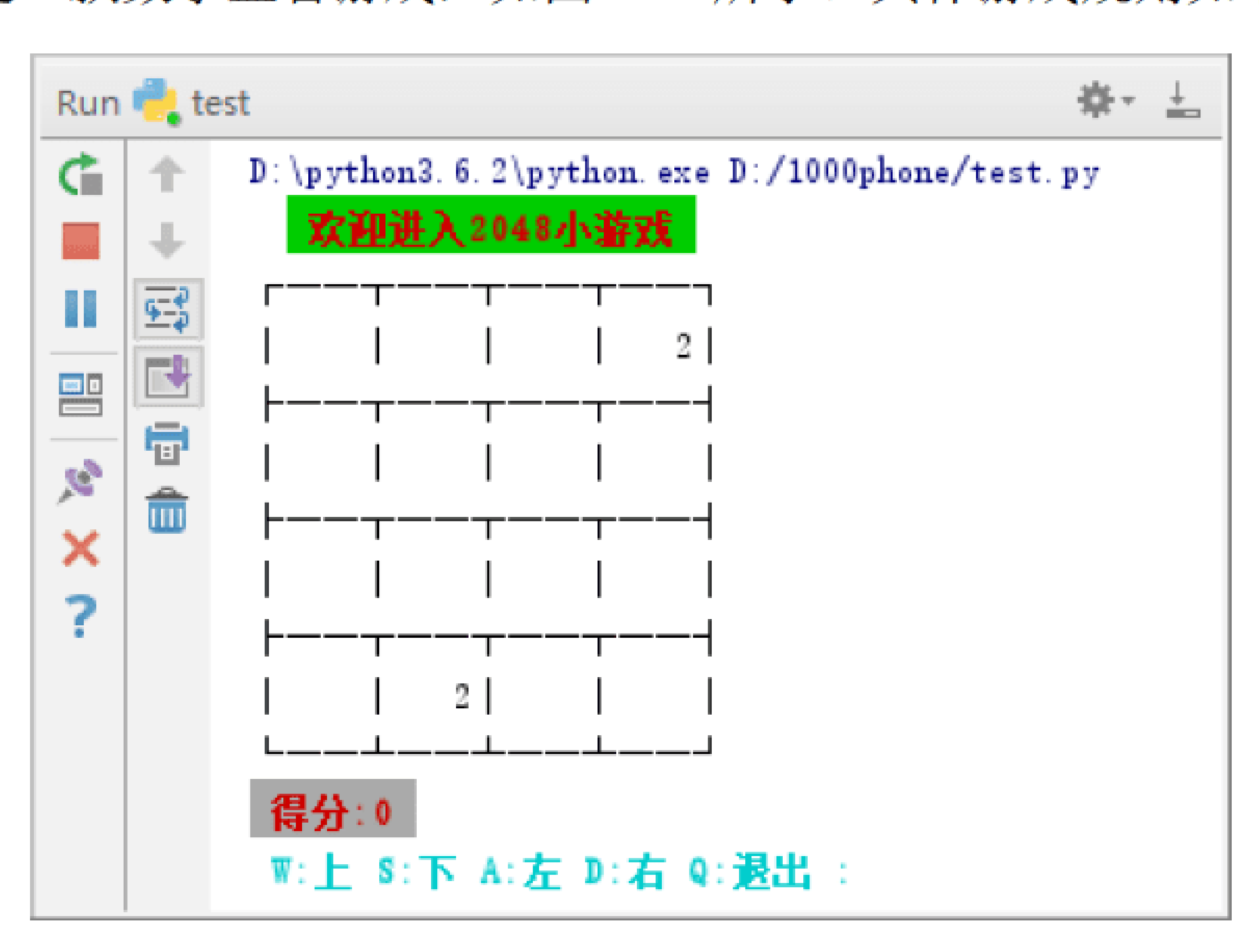


图 14.1 2048 游戏界面

① 玩家每次可以选择上下左右其中一个方向移动。

② 每移动一次，所有数字方块都会往移动的方向靠拢。

③ 相同数字方块在靠拢时会相加。

④ 每次移动完成后，系统会在空白的方块中随机添加 2 或 4。

⑤ 当所有方块中填满数字并不能相加时，游戏结束。

⑥ 玩家的得分为相同数字之和的累加。

根据上述游戏规则，该游戏须实现以下功能：

① 显示游戏界面。

② 上下左右移动。

③ 添加随机数字。

④ 判断游戏是否结束。

为方便读者理解各功能之间的联系，此处画出程序的流程图，如图 14.2 所示。

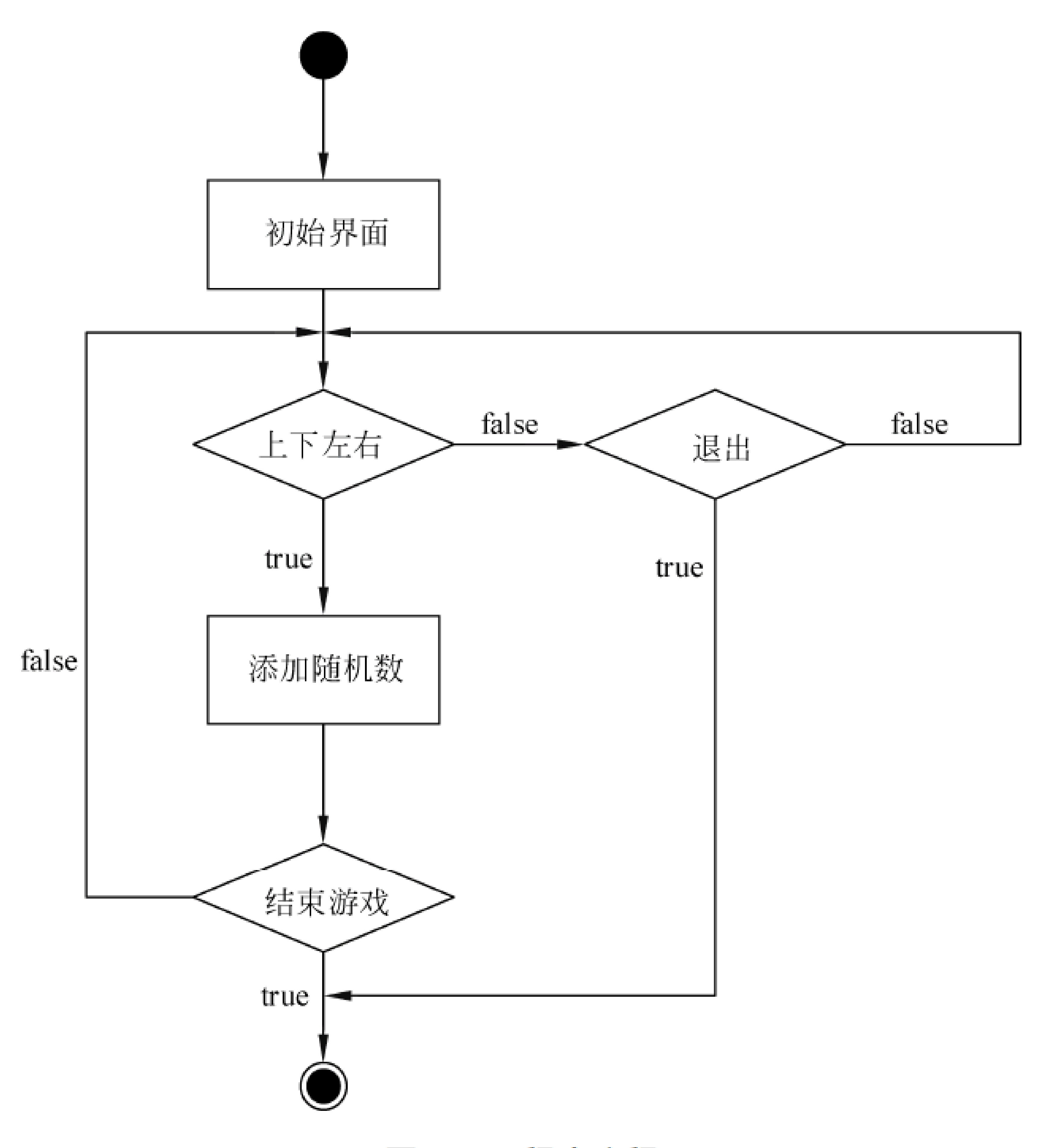


图 14.2　程序流程

14.2　程序设计

对程序各功能有了初步了解后，本节将带领大家实现每个功能。首先，程序须选用合适的数据结构，由于游戏可以看成由 4×4 个数字组成，每移动一次，就是对这 4×4 个数字进行操作，因此数据结构可以选择二维列表（二维列表类似于二维矩阵），具体如下所示：

```
matix = [[0 for i in range(4)] for i in range(4)]  # 初始化生成一个二维列表
score = 0 # score 记录游戏的分数
```

通过 print()函数打印该二维列表，则输出结果如下所示：

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

二维列表中的数据须显示在 4×4 方块中，具体实现如下所示：

```
1   def display():
2       print('  \033[1;31;42m 欢迎进入 2048 小游戏 \033[0m')
3       print('\r'
4             '┌────┬────┬────┬────┐\n'
5             '│%4s│%4s│%4s│%4s│\n'
6             '├────┼────┼────┼────┤\n'
7             '│%4s│%4s│%4s│%4s│\n'
8             '├────┼────┼────┼────┤\n'
9             '│%4s│%4s│%4s│%4s│\n'
10            '├────┼────┼────┼────┤\n'
11            '│%4s│%4s│%4s│%4s│\n'
12            '└────┴────┴────┴────┘'
13            % (ifZero(matix[0][0]), ifZero(matix[0][1]),
14               ifZero(matix[0][2]), ifZero(matix[0][3]),
15               ifZero(matix[1][0]), ifZero(matix[1][1]),
16               ifZero(matix[1][2]), ifZero(matix[1][3]),
17               ifZero(matix[2][0]), ifZero(matix[2][1]),
18               ifZero(matix[2][2]), ifZero(matix[2][3]),
19               ifZero(matix[3][0]), ifZero(matix[3][1]),
20               ifZero(matix[3][2]), ifZero(matix[3][3]),)
21            )
22      print('\033[1;31;47m 得分:%s \033[0m' % (score))
```

其中，第 2 行与第 22 行输出带颜色的字符串，其语法格式如下：

```
\033[显示方式;前景色;背景色 m
```

显示方式、前景色与背景色都用数字表示，具体如表 14.1 与表 14.2 所示。

表 14.1　显示方式

显 示 方 式	说　明
0	终端默认设置
1	高亮显示
4	使用下画线
5	闪烁
7	反白可见
8	不可见

ifZero ()函数的实现如下所示：

```
def ifZero(s):
```

```
    return s if s != 0 else ''
```

表 14.2 前景色与背景色

前景色	背景色	说明
30	40	黑色
31	41	红色
32	42	绿色
33	43	黄色
34	44	蓝色
35	45	紫红色
36	46	青蓝色
37	47	白色

其中，若参数 s 为 0，则返回空字符，否则返回参数 s。

此时，只需在上述界面的基础上再随机生成两个数（2 或 4）就可以构成初始界面，具体如下所示：

```
def init():
    initNumFlag = 0
    while True:
        k = 2 if random.randrange(0, 10) > 1 else 4  # 随机生成 2 或 4
        s = divmod(random.randrange(0, 16), 4)  # 生成矩阵初始化的下标
        if matix[s[0]][s[1]] == 0: # 只有当其值不为 0 时才赋值,避免第二个值重复
            matix[s[0]][s[1]] = k
            initNumFlag += 1
            if initNumFlag == 2:
                break
    display()
```

程序调用 init()函数生成初始界面，如图 14.3 所示。

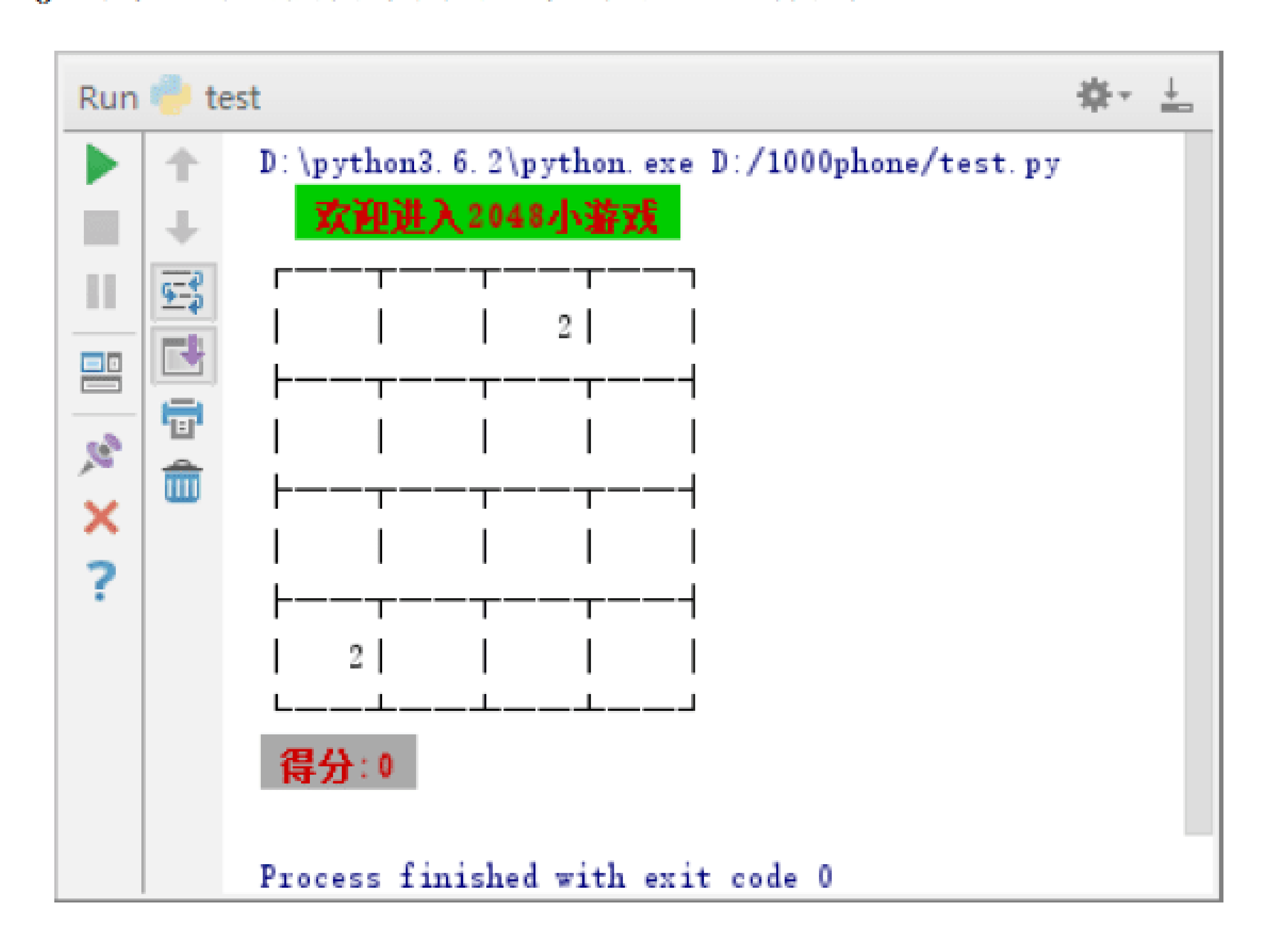


图 14.3 初始界面

界面显示功能完成后，接下来实现上下左右功能，它是整个游戏的核心功能。由于上下左右 4 个方向的具体实现代码类似，此处只讲解向右移动的实现，具体如下所示：

```
1   def moveRight():
2       global score
3       for i in range(4):
4           for j in range(3, 0, -1):
5               for k in range(j - 1, -1, -1):
6                   if matix[i][k] > 0:
7                       if matix[i][j] == 0:
8                           if k > 0 and matix[i][k] == matix[i][k - 1]:
9                               matix[i][k] *= 2
10                              score += matix[i][k]  # 将当前数作为 score 加上
11                              matix[i][k - 1] = 0
12                          if k == 2 and matix[i][k - 1] == 0 and \
13                                      matix[i][k] == matix[i][k - 2]:
14                              matix[i][k] *= 2
15                              score += matix[i][k]
16                              matix[i][k - 2] = 0
17                          matix[i][j] = matix[i][k]
18                          matix[i][k] = 0
19                      elif matix[i][j] == matix[i][k]:
20                          matix[i][j] *= 2
21                          score += matix[i][j]
22                          matix[i][k] = 0
23                      break
```

其中，i 控制行，j 控制列，k 控制第 i 行的前 j–1 个数字，注意 i 与 j 都是从 0 开始。第 8～11 行代码用于处理图 14.4（假设 j = 3、k = 2）中出现的情形，第 12～16 行代码用于处理图 14.5（假设 j = 3、k = 2）中出现的情形，第 19～22 行用于处理图 14.6（假设 j = 3、k = 2）中出现的情形。

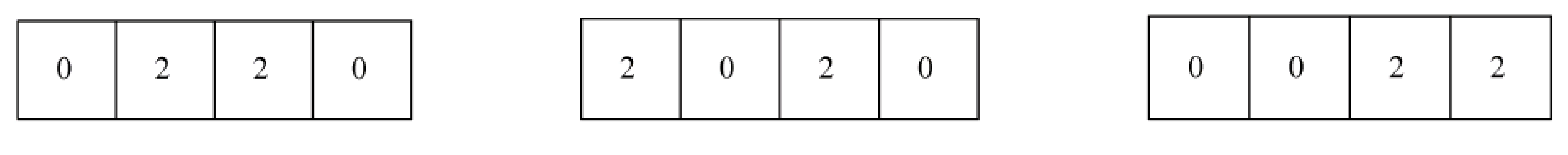


图 14.4　某行数字（一）　**图 14.5　某行数字（二）**　**图 14.6　某行数字（三）**

上述代码就可以实现右移的功能，大家可以根据此代码自己实现左移、上移、下移功能，在此不再赘述。

每次执行移动操作后，程序中须自动添加一个随机数（2 或 4）并重新输出显示界面，具体如下所示：

```
def addRandomNum():
    while True:
        k = 2 if random.randrange(0, 10) > 1 else 4
```

```
        s = divmod(random.randrange(0, 16), 4)
        if matix[s[0]][s[1]] == 0:
            matix[s[0]][s[1]] = k
            break
    display()
```

此处只需将随机数添加到对应矩阵不为 0 的元素处，再调用 display()函数显示界面即可。

最后，程序须检查游戏是否结束，具体如下所示：

```
def checkOver():
    for i in range(4):
        for j in range(3):
            if matix[i][j] == 0 or matix[i][j] == matix[i][j + 1] or \
                                matix[j][i] == matix[j + 1][i]:
                return True
    else:
        return False
```

上述代码使用 for 循环遍历二维矩阵中每个元素，若存在某个元素为 0 或存在某个元素可以与周围的元素相加时，游戏未结束；否则，游戏结束。

14.3 代码实现

14.2 节介绍了 2048 游戏中必须实现的各种功能，接下来讲解如何将这些功能组合成程序流程，具体如下所示：

```
def main():
    flag = True
    init()
    while flag: # 循环标志
        d = input('\033[1;36;1m W:上 S:下 A:左 D:右 Q:退出 :\033[0m')
        if d == 'A':      # 左移
            moveLeft()
        elif d == 'S':    # 下移
            moveDown()
        elif d == 'W':    # 上移
            moveUp()
        elif d == 'D':    # 右移
            moveRight()
        elif d == 'Q':    # 退出
            break
        else:          # 对用户的其他输入不处理
```

```
            continue
        addRandomNum()
        if not checkOver():
            print('游戏结束')
            flag = False
```

上述代码使用 while 循环与 if-elif-else 语句来控制整个程序的流程。

接下来编写代码，测试整个程序，具体如下所示：

```
if __name__ == '__main__':
    main()
```

至此，整个程序编写完成。

14.4 效 果 演 示

程序代码编辑完成，如果没有错误，便可运行。本节演示程序运行效果。

1．初始界面

程序运行后，首先进入初始界面，如图 14.7 所示。

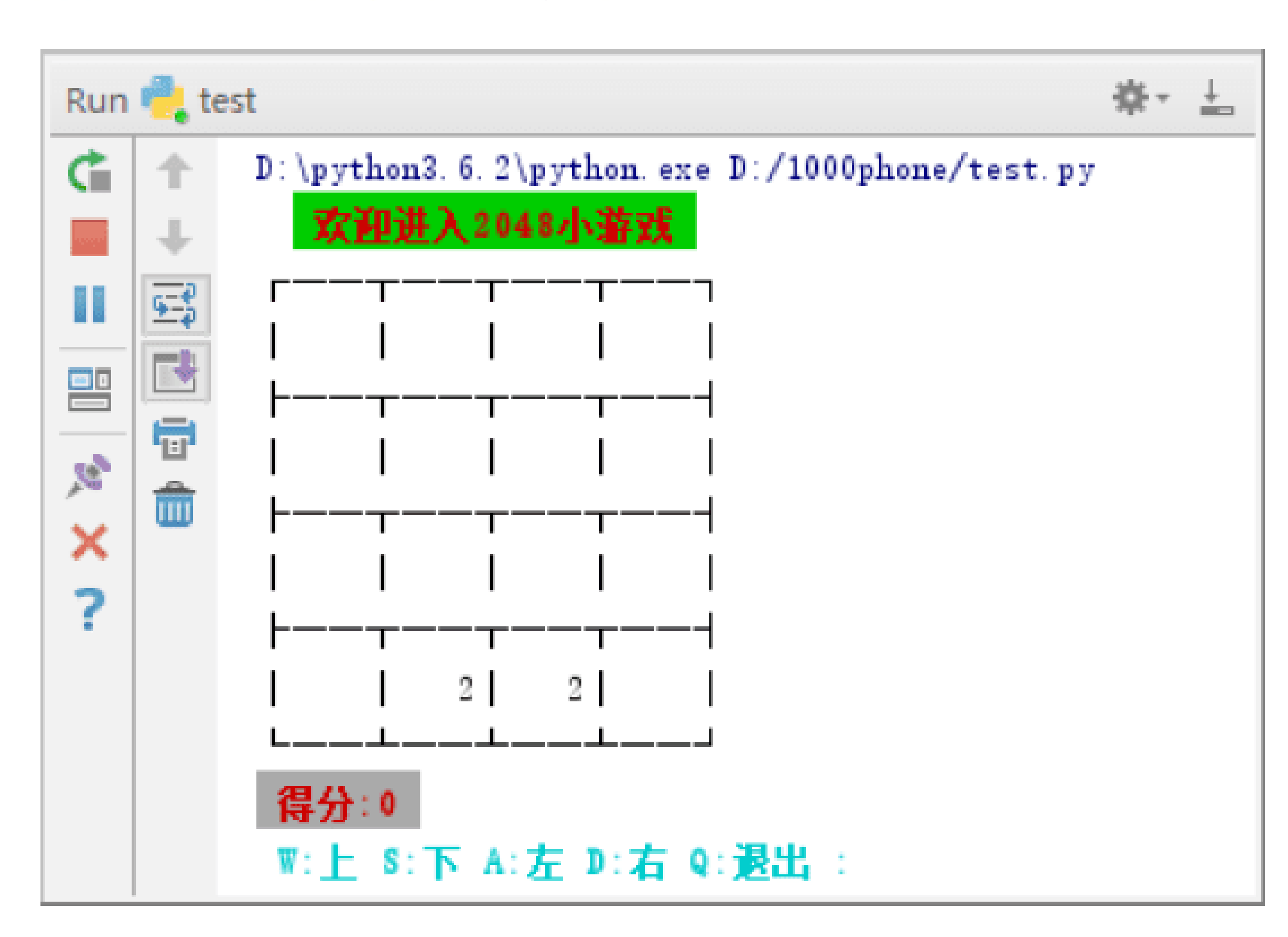


图 14.7 初始界面

2．上移

当输入 W 时，游戏中所有数字会向上移动。移动过程中，相同数字方块在靠拢时会相加，并将结果与得分相加。移动完成后，程序会随机添加一个数字，如图 14.8 所示。

3．下移

当输入 S 时，游戏中所有数字会向下移动。移动过程中，相同数字方块在靠拢时会相加，并将结果与得分相加。移动完成后，程序会随机添加一个数字，如图 14.9 所示。

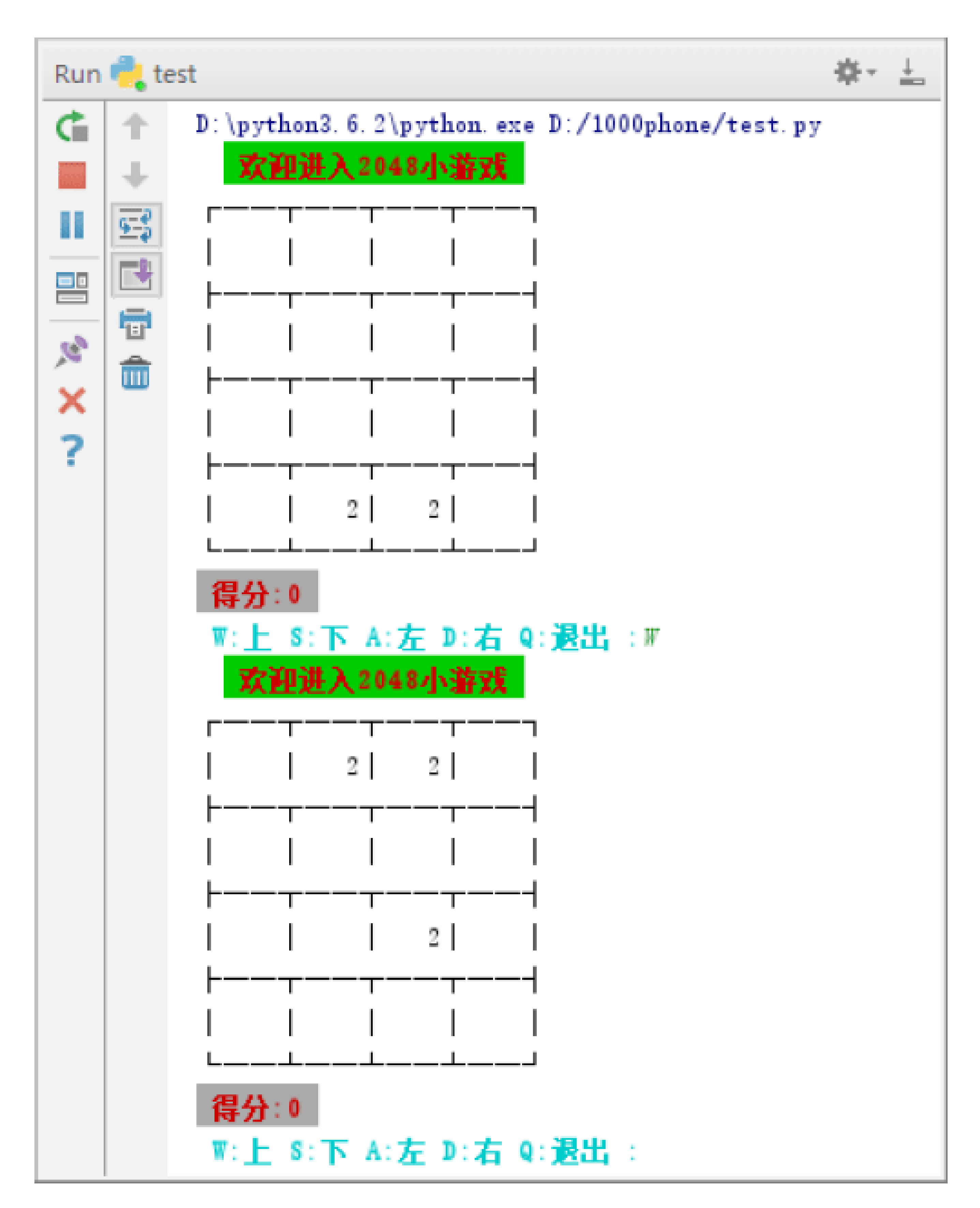


图 14.8　上移界面

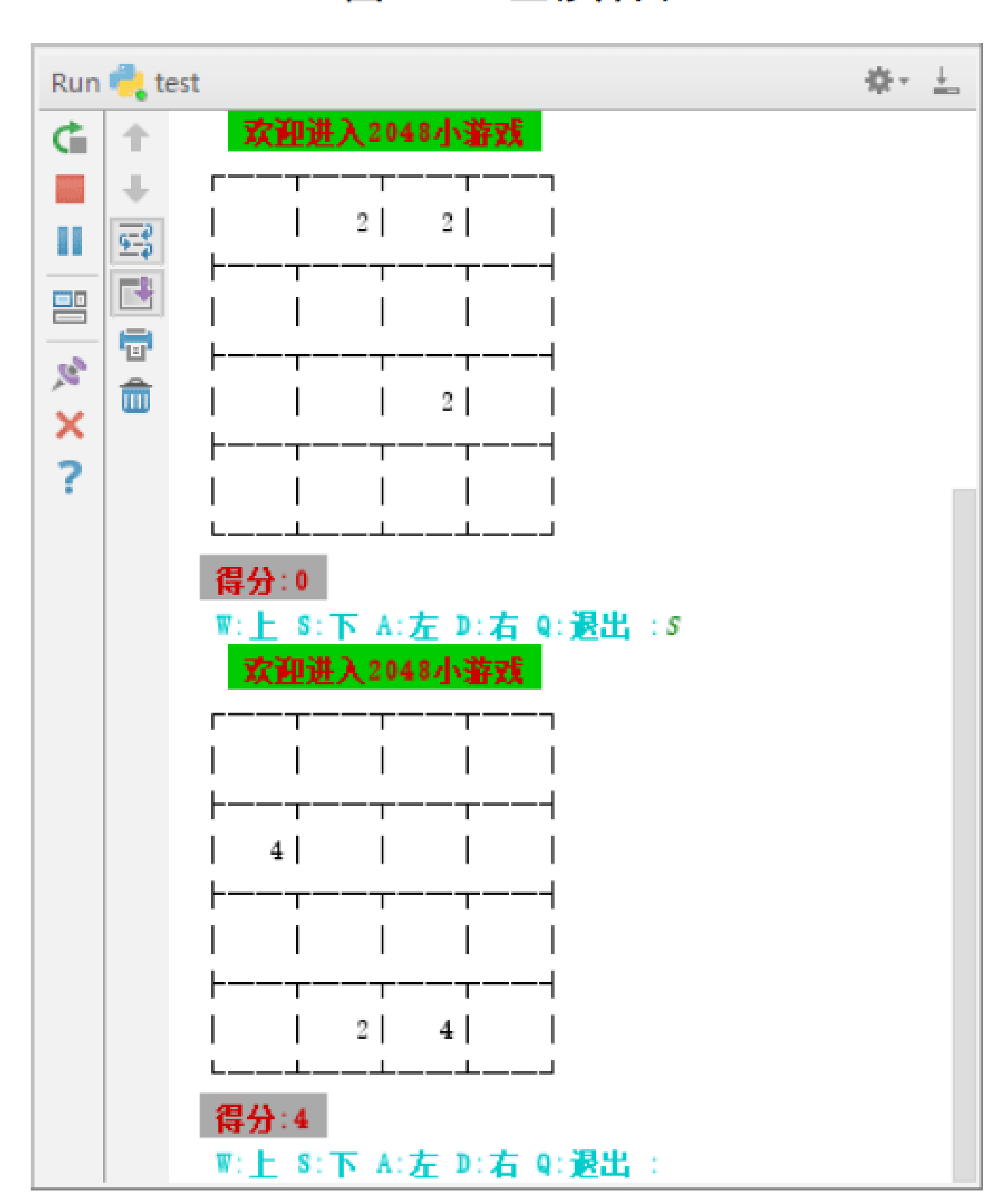


图 14.9　下移界面

4．左移

当输入 A 时，游戏中所有数字会向左移动。移动过程中，相同数字方块在靠拢时会

相加，并将结果与得分相加。移动完成后，程序会随机添加一个数字，如图 14.10 所示。

Run test
欢迎进入2048小游戏
得分:4
W:上 S:下 A:左 D:右 Q:退出 :A
欢迎进入2048小游戏
得分:4
W:上 S:下 A:左 D:右 Q:退出 :

图 14.10　左移界面

5．右移

当输入 D 时，游戏中所有数字会向右移动。移动过程中，相同数字方块在靠拢时会相加，并将结果与得分相加。移动完成后，程序会随机添加一个数字，如图 14.11 所示。

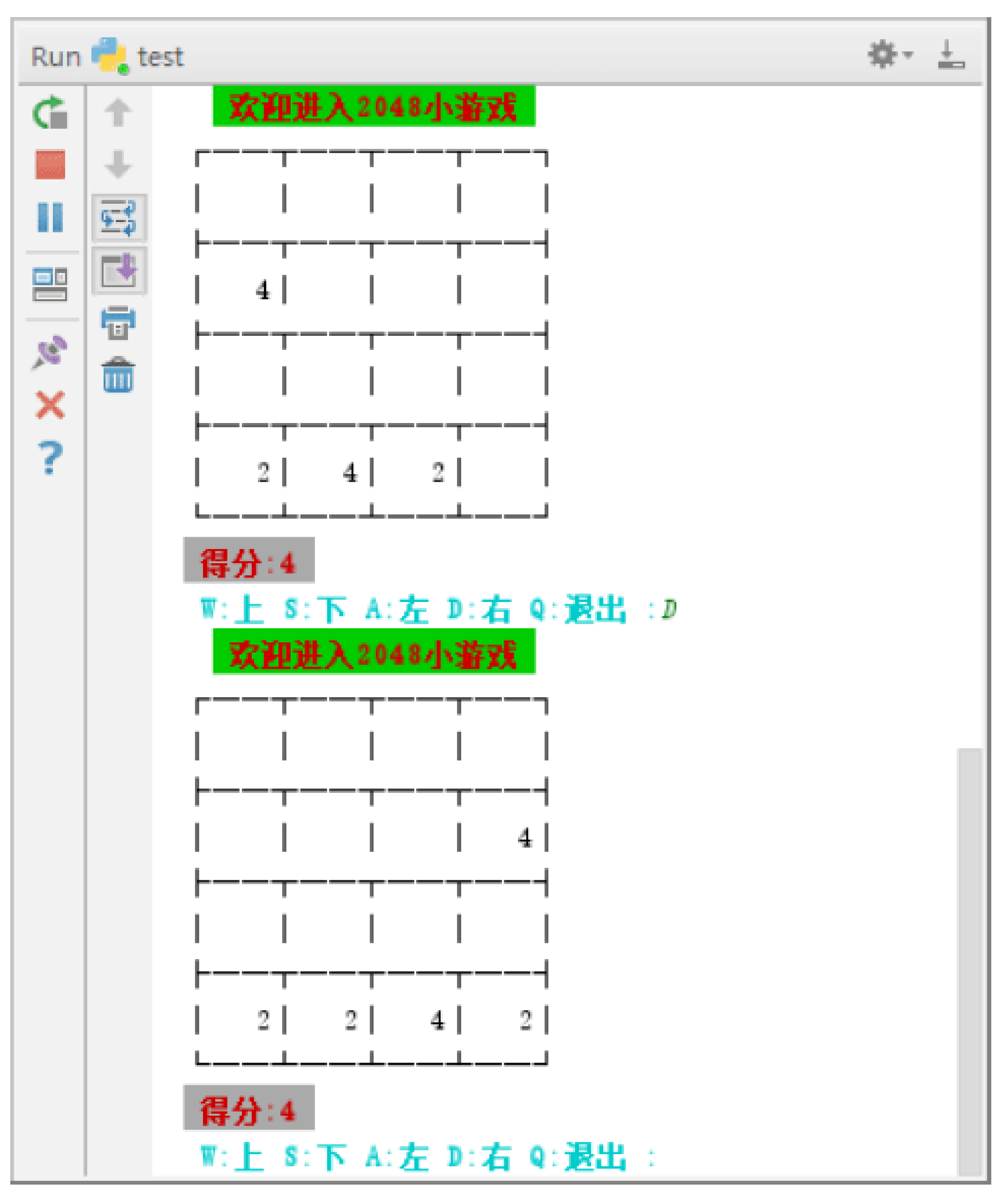


图 14.11　右移界面

6. 退出游戏

当输入 Q 时，程序退出游戏，如图 14.12 所示。

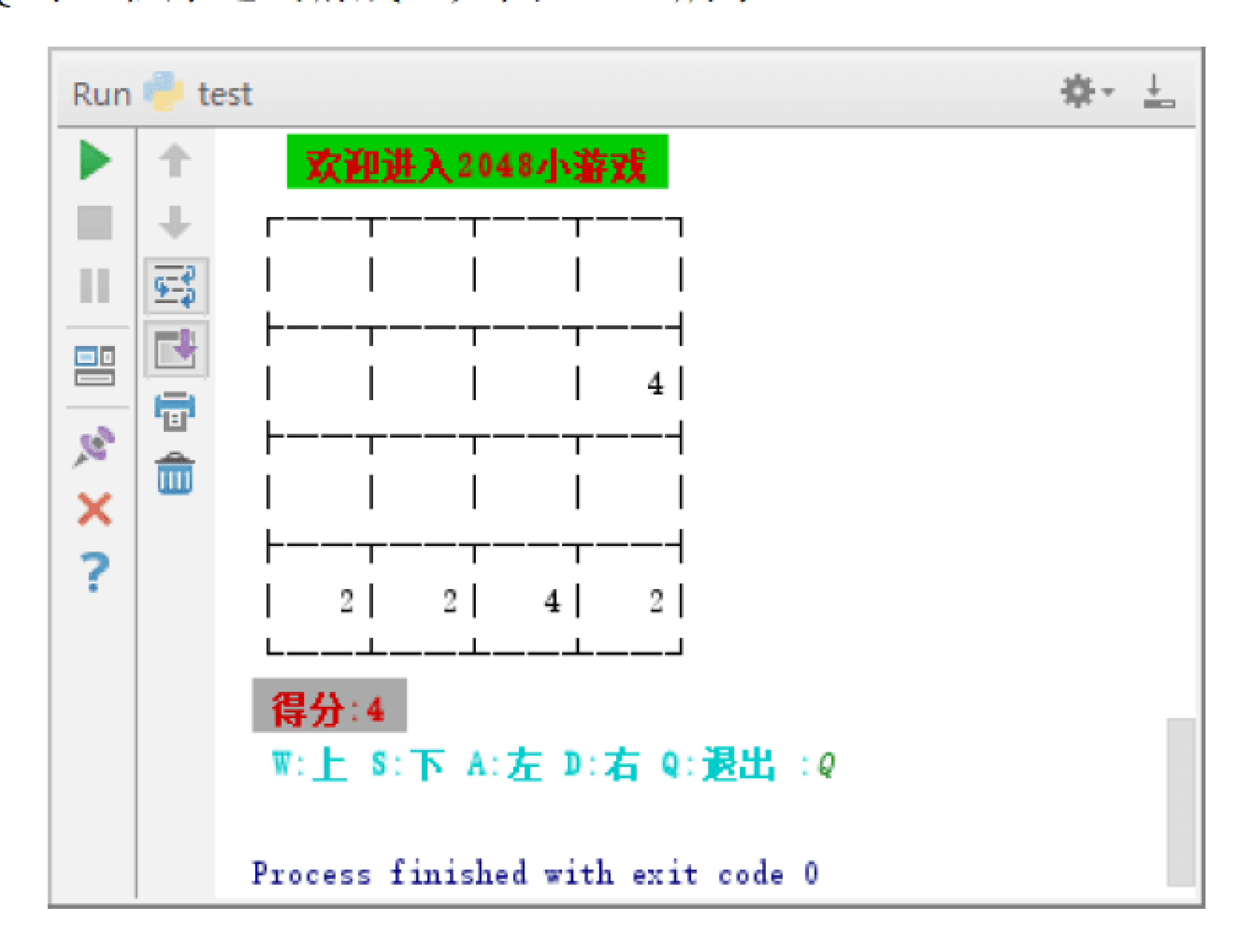


图 14.12 退出游戏

14.5 本章小结

通过本章的学习，大家须掌握 Python 语言的开发流程和技巧，熟练运用 Python 语言基础知识，提高运用 Python 语言解决实际问题的能力。

14.6 课外实践

尝试用面向对象思想实现 2048 游戏。



附录 A

常用模块和内置函数操作指南

基本的标准模块如表 A.1 所示。

表 A.1 基本的标准模块

模 块 名	说 明
math	数学函数
os	处理文件或目录
random	生成随机数
re	正则匹配
time	处理时间或日期
sqlite3	创建并使用 SQLite 数据库
sys	Python 解释器的设置
itertools	操作迭代对象

读写文件模块如表 A.2 所示。

表 A.2 读写文件模块

模 块 名	说 明
csv	读写 CSV（Comma-Separated Values，字符分隔值）文件
json	处理 JSON 文件
xml	分析 XML 文件
zipfile	读写.zip 文件

数据分析模块如表 A.3 所示。

表 A.3 数据分析模块

模 块 名	说 明
numpy	快速运算矩阵
json	Python 对象与 JSON 字符串之间相互转换
pandas	分析表格数据
scipy	科学计算
scikit-learn	机器学习

数据可视化模块如表 A.4 所示。

表 A.4 数据可视化模块

模 块 名	说 明
matplotlib	绘制图表
image manipulation	处理图像
bokeh	交互式绘图

与网络相关的模块如表 A.5 所示。

表 A.5 与网络相关的模块

模 块 名	说 明
requests	获取网页
BeautifulSoup	解析 HTML 页面
paramiko	通过 SSH 执行命令

常用内置函数如表 A.6 所示。

表 A.6 常用内置函数

函 数	说 明
abs(x)	求 x 的绝对值
all(iterable)	接受一个迭代器，如果迭代器的所有元素都为真，那么返回 True，否则返回 False
any(iterable)	接受一个迭代器，如果迭代器里有一个元素为真，那么返回 True，否则返回 False
bin(x)	将整数 x 转换为二进制字符串
bool([x])	将 x 转换为布尔类型
bytes([source[, encoding[, errors]]])	将一个字符串转换成字节类型
callable(object)	判断对象是否可以被调用
chr(i)	返回值是当前整数对应的 ASCII 字符
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=–1)	将字符串编译成 Python 能识别或可以执行的代码，也可以将文字读成字符串再编译
complex([real[, imag]])	创建一个复数
divmod(a,b)	返回一个元组(a // b, a % b)
enumerate(iterable, start=0)	返回一个可以枚举的对象
eval(expression, globals=None, locals=None)	将字符串当成有效的表达式来求值并返回计算结果
exec(object[, globals[, locals]])	执行字符串或 compile()方法编译过的字符串
filter(function, iterable)	用于过滤序列，过滤掉不符合条件的元素，返回由符合条件元素组成的新列表
float([x])	将整数和字符串转换成浮点数
format(value[,format_spec])	格式化输出字符串，格式化的参数顺序从 0 开始
frozenset([iterable])	创建不可变集合
getattr(object, name[, default])	获取对象的属性
globals()	返回一个描述当前全局变量的字典
hasattr(object, name)	用于判断对象是否包含对应的属性
hash(object)	用于获取一个对象（字符串或者数值等）的哈希值

续表

函　数	说　明
help([object])	返回对象的帮助文档
hex(x)	将整数 x 转换成十六进制字符串
id(object)	返回对象的内存地址
input([prompt])	获取用户输入内容
int([x[,base=10]])	将一个数字或 base 类型的字符串转换成整数
isinstance(object, classinfo)	检查对象是否是类的对象
issubclass(class, classinfo)	检查一个类是否是另一个类的子类
iter(object[, sentinel])	用来生成迭代器
len(s)	返回对象（字符、列表、元组等）长度或项目个数
list([iterable])	列表构造函数
locals()	返回当前可用的局部变量的字典
map(function, iterable, …)	根据提供的函数对指定序列做映射
max(arg1, arg2, *args[, key])	返回给定参数的最大值
memoryview(obj)	返回给定参数的内存查看对象
min(arg1, arg2, *args[, key])	返回给定参数的最小值
next(iterator[, default])	返回一个可迭代数据结构中的下一项
oct(x)	将整数 x 转换成八进制字符串
open(file, mode='r', buffering=−1, encoding=None, errors=None, newline=None, closefd=True, opener=None)	用于打开一个文件，创建一个 file 对象，相关的方法才可以调用它进行读写
ord(c)	返回对应的十进制整数
pow(x, y[, z])	计算 x 的 y 次方，如果 z 已存在，则对结果进行取模
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)	用于打印输出
range([start,]stop[,step])	返回一个可迭代对象，从 start （默认为 0）开始，到 stop 结束（不包括 stop），step 为步长（默认为 1）
repr(object)	将对象转化为供解释器读取的形式
reversed(seq)	反转列表中元素
round(x[,n])	对浮点数进行近似取值
set([iterable])	创建一个无序不重复元素集
setattr(object, name, value)	用于设置属性值
slice(start, stop[, step])	实现切片功能
sorted(iterable, key=None, reverse=False)	对所有可迭代的对象进行排序操作
str(object='')	转换为字符串类型
sum(iterable[, start])	进行求和计算
super([type[, object-or-type]])	用于调用父类的一个方法
tuple([iterable])	将序列转化为元组
type(object)	返回对象所属的类型
vars([object])	返回对象的属性和属性值的字典对象
zip(*iterables)	用于将可迭代对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的资源，有需求的读者请扫描下方的“书圈”微信公众号二维码，在图书专区下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京市海淀区双清路学研大厦 A 座 701

邮　　编：100084

电　　话：010－62770175－4608

资源下载：http://www.tup.com.cn

客服邮箱：tupjsj@vip.163.com

QQ：2301891038（请写明您的单位和姓名）

资源下载、样书申请



书圈



扫一扫，获取最新目录

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。